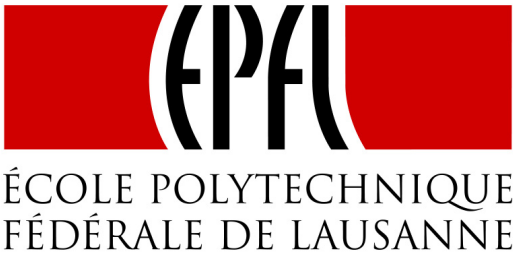


Introducing dynamicity in JavaBIP

Valentin Rutz

This thesis was prepared under the supervision of
Prof. Joseph Sifakis,
Anastasia Mavridou
and
Dr. Simon Bliudze

RiSD laboratory - IC section
École Polytechnique Fédérale de Lausanne
Lausanne, Switzerland
June 2016

Contents

1 Introduction **4**
- 1.1 BIP . . . 4
- 1.2 JavaBIP . . . 5

2 Overview of static JavaBIP **7**
- 2.1 System specification . . . 8
  - 2.1.1 Behaviour specification . . . 8
  - 2.1.2 Glue specification . . . 9
  - 2.1.3 Data specification . . . 10
- 2.2 JavaBIP module . . . 10
- 2.3 JavaBIP engine . . . 11
  - 2.3.1 Engine kernel . . . 11
  - 2.3.2 Coordinators . . . 11
  - 2.3.3 Encoders . . . 12

3 Defining dynamicity in JavaBIP **13**
- 3.1 Dynamicity . . . 13
  - 3.1.1 Implementation challenges . . . 13
  - 3.1.2 Dynamicity mechanisms . . . 14
- 3.2 Validity . . . 14
  - 3.2.1 Definition of system validity . . . 14
  - 3.2.2 Determining system validity . . . 15
    - Adding a component . . . 15
    - Removing a component . . . 15
    - Determining if the system is valid . . . 15
  - 3.2.3 Validity proofs . . . 16
  - 3.2.4 Examples of validity graphs . . . 17
- 3.3 Boolean encoding using BDDs . . . 20
  - 3.3.1 Behaviour encoding . . . 21
  - 3.3.2 Current state encoding . . . 21
  - 3.3.3 Glue encoding . . . 22
  - 3.3.4 Data encoding . . . 23

4 Implementation **25**
4.1 ComponentPool . . . 26
4.2 The BIPEngine interface . . . 26
4.2.1 Starting and executing the engine . . . 27
4.2.2 Initializing the engine . . . 28
4.2.3 Registering a new component . . . 28
4.2.4 Informing the engine . . . 29
4.2.5 Stopping the engine . . . 30

5 Performance **31**
5.1 Registration . . . 31
5.1.1 Registration before engine start . . . 31
Without data . . . 31
With data . . . 31
5.1.2 Registration with engine start . . . 33
Without data . . . 33
With data . . . 33
5.1.3 Registration on-the-fly . . . 33
Without data . . . 33
With data . . . 33
5.2 Cycles . . . 33

6 Future work **34**

7 Conclusion **36**

Chapter 1

Introduction

Concurrency requires developers to be very meticulous with the primitives they use. Most mainstream programming languages like Java, C and C++ have low level primitives such as locks, semaphores which are mixed up with the functional code. This results in code code that is difficult to read, modify and maintain. Using low level synchronization primitives requires a great attention to every possible case that could lead to deadlocks, race conditions, etc. This is an important reason why concurrency programming is very error-prone, especially when building large systems.

The actor model [1] is a solution to the aforementioned problem. Actors do not have access to the states of other actors and communicate using pure asynchronous message passing but unfortunately, it can be not enough. For example, when a protocol must be enforced, such as an order in the messages received by different actors, then some synchronization is unavoidable.

1.1 BIP

JavaBIP is a Java variation of the BIP framework [2] for the component-based design of correct-by-construction concurrent applications. It provides a simple, but powerful coordination mechanism by superposing three layers: Behavior, Interaction, and Priority. The first layer describes the behavior of components as FSMs having transitions labeled with ports and extended with data stored in local variables. Ports form the interface of a component and are used to define its interactions with other components. They can also export part of the local variables, allowing access to the component's data. The second layer defines component coordination by means of *interaction models*, i.e., sets of interactions. Interactions are sets of ports that define allowed synchronizations between components. An interaction model is defined in a structured manner by using connectors [3]. For each interaction, a connector also specifies how the data is retrieved, filtered and updated in each of the participating components. In particular, a Boolean guard can be associated to an interaction. The interaction is only enabled if the data provided by the components satisfies the guard [4]. In the third layer, priorities are used to impose scheduling constraints and to resolve conflicts when multiple interactions are enabled

simultaneously. 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 an interaction satisfying the glue specification, performs the data transfer and notifies all the involved components;
3. The notified components execute the functions associated with the corresponding transitions.

1.2 JavaBIP

JavaBIP [5, 6] is an adaptation in Java of BIP. JavaBIP can be viewed as a generalisation of the actor model since it provides two mechanisms for component interaction:

1. Synchronization of transitions of several components;
2. Asynchronous event notifications.

The latter can be used to emulate asynchronous messages.

It allows developers to think on a high level of abstraction by separating functional code from the coordination aspects of the system behaviour. Coordination is done in an exogenous manner, relying totally on component APIs and external specifications.

JavaBIP raises the level of abstraction and provides clear separation of concerns between the behaviour of components (FSMs) and their coordination, i.e. synchronisations defined in the *Glue*. This separation allows developers to write more complex concurrent system with more readable and easier to debug code. JavaBIP lets one focus on the functional behaviour of the program by providing a simple primitive mechanism for synchronisation of transitions between multiple components.

JavaBIP allows writing large concurrent systems but it does have limitations as well. It is a static framework, which means that everything needs to be set in place before launching the system. The slightest modification of the program structure in terms of components and connections among them requires stopping the system, applying the required modifications and relaunching it. Doing so means losing all work done so far and it is not optimal to shutdown a large system for a slight modification of a small part of said system.

The implementation of JavaBIP presented in Section 2 is static. This means that it allows coordinating only software components that are statically defined prior to system deployment. Modern systems, including large banking systems or systems designed for

modular smartphones (Google Ara project) make use of components that can register and unregister during the system execution.

The Camel Routes system is a very pertinent example of this limitation. Camel routes are extensively utilized in Connectivity Factory ™ - the flagship product of Crossing-Tech S.A [7]. 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 would imagine that if we have a monitor in the system, the routes would come and go exchanging information with the monitor to know if they can use resources or not. We can coordinate (e.g. limit the number of) Camel Routes with the static implementation of JavaBIP. To do so, we need to register all routes of the system during the initialization of the JavaBIP engine. Nevertheless, if a new route is created we would need to stop the JavaBIP engine and re-register all routes, including the new one, to continue coordinating the routes. Whereas the dynamic implementation of JavaBIP would allow us to skip the part where we stop the JavaBIP engine. So we would not need those superfluous steps and we would be able to add this new route on-the-fly.

This thesis presents how JavaBIP was extended to handle this type of dynamicity. In particular, we present the theoretical foundations and the implementation of our approach.

The structure of this thesis is as follows: Section 2 presents the static implementation of JavaBIP. Section 3 explains the dynamic implementation of JavaBIP with its features, challenges and some modifications that had to be done to accommodate dynamicity. Section 4 explains the implementation in more details. Section 5 presents the performance evaluation. Section 6 discusses some future work such as dynamic addition of types, registering batches of components, etc.

Chapter 2

Overview of static JavaBIP

In this chapter, we describe the static implementation of the JavaBIP framework. For a graphical overview of the software architecture of JavaBIP, refer to Figure 2.1.

JavaBIP consists of two main parts: the modules and the engine, as shown, respectively, in the top and bottom parts of the figure. The exchange of information is exchanged among the engine and the modules is illustrated in Figure 2.1 with arrows. Information is either exchanged only once at initialization of the engine (illustrated in Figure 2.1 with continuous arrows) or at each execution cycle (illustrated in Figure 2.1 with dashed arrows).

For instance, each module sends behaviour, glue and data-wire specifications to the engine:

- Behaviour specification for each component, given by an FSM extended with ports and data. This is provided as an annotated Java class, whereof the methods can call APIs provided by the coordinated components.
- Glue specification, which is the interaction model of the system that specifies how the transitions of different components must be synchronized. This is provided as an XML configuration file.
- Optionally, data transfer can be defined, by providing the data-wire specification for each data variable of every component, that specifies which data are exchanged between components. This is provided as an XML configuration file.

The specifications are categorized either as permanent or temporary, illustrated in Figure 2.1 with arrows labelled permanent and temporary, respectively. Permanent specifications are sent only at initialization of the engine, while temporary specifications are sent at each execution cycle.

The implementation of the engine is modular. It consists of a stack of coordinators and the kernel. The coordinators manage the flow of information between the modules and the kernel. Coordinators use dedicated encoders to transform the specifications acquired from the modules into permanent and temporary constraints that are sent to the kernel.

The kernel solves the combined constraints imposed by the behavior, glue and data-wire specifications and passes the solution back to the coordinators. Each coordinator interprets the relevant part of the solution and triggers the corresponding action in the executors, where the actual API function calls to the controlled source code are made. How the solution is forwarded from the kernel to the functional code is illustrated in Figure 2.1 with dashed arrows labelled execute. If the kernel cannot find a solution because the combined constraints are contradictory, a deadlock occurs.

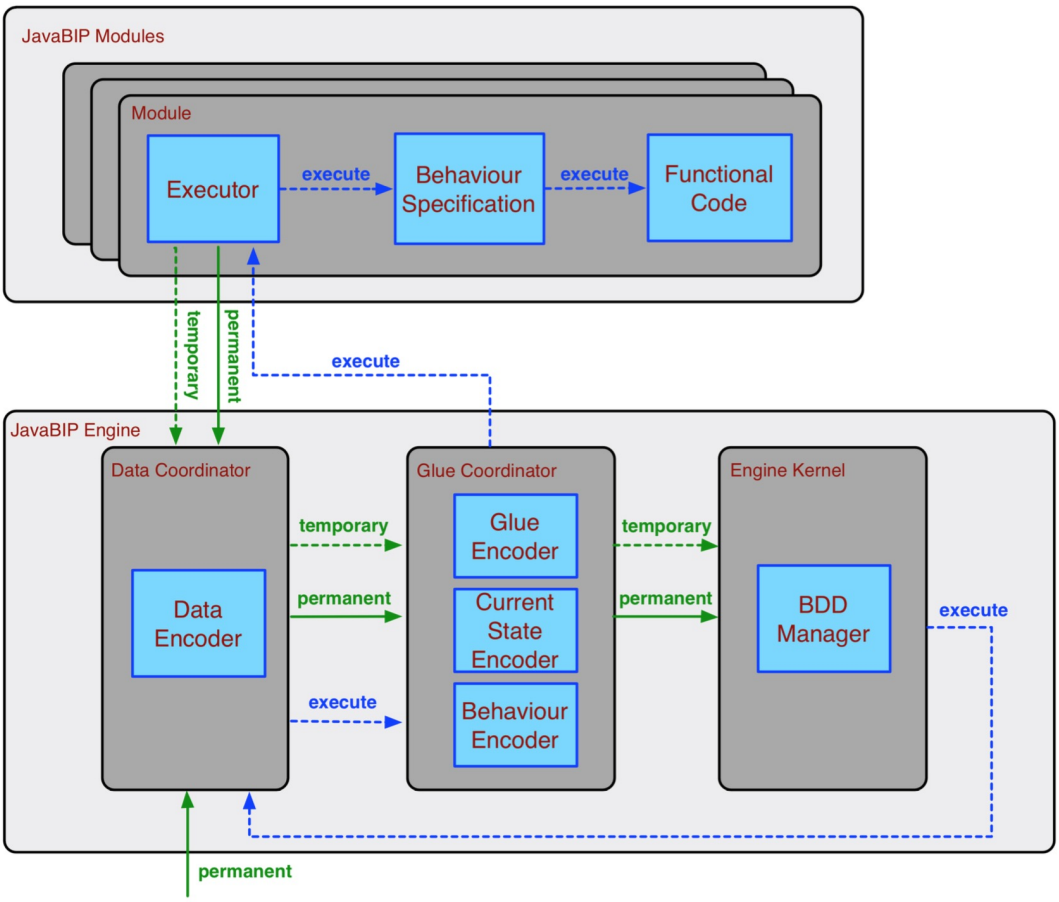


Figure 2.1: JavaBIP runnable system software architecture

2.1 System specification

2.1.1 Behaviour specification

The behaviour specification is given using annotated classes. In the example below, we can see that all ports are described using the `@Port` annotation, methods are linked to transitions using the `@Transition` annotation, etc.

Listing 2.1: Behaviour: annotated class in Java

```
1  @Ports({ @Port(name = "log", type = PortType.enforceable),
2          @Port(name = "broadcast", type = PortType.enforceable) })
3  @ComponentType(initial = "zero", name = "org.bip.spec.Tracker")
4  public class Tracker {
5
6      public int noOfTransitions = 0;
7
8      Logger logger = LoggerFactory.getLogger(Tracker.class);
9      private int trackerId;
10
11     public Tracker(int id) {
12         trackerId = id;
13     }
14
15     @Transition(name = "log", source = "zero", target = "zero")
16     public void logging() {
17         // System.out.println("Peer has updates his status");
18         noOfTransitions++;
19     }
20
21     @Transition(name = "broadcast", source = "zero", target = "zero")
22     public void broadcasting() {
23         noOfTransitions++;
24         // System.out.println("Broadcasting " + trackerId);
25     }
26
27     @Data(name = "trackerId", accessTypePort = AccessType.any)
28     public int trackerId() {
29         return trackerId;
30     }
31
32 }
```

2.1.2 Glue specification

Interaction constraints are specified using macro-notation:

- Causal constraints (Require) specify ports of other components, necessary for any interaction involving the port with which the constraint is associated.
- Acceptance constraints (Accept) define optional ports of other components, accepted in the interactions involving the port with which the constraint is associated.

The Glue specification must be provided in an XML file as seen below. Each constraint has two parts: effect and causes. The former defines the port to which the constraint is associated intuitively, the effect is the firing of a transition labeled by this port. The latter lists the ports that are necessary to "cause" the "effect". For the require constraints, all causes must be present. For the accept constraints, any (possibly empty) combination of the causes is accepted.

Listing 2.2: Glue: examples of require and accept clauses

```
<require>
    <effect id="register" specType="org.bip.spec.Peer"/>
    <causes>
        <option>
            <causes>
                <port id="log" specType="org.bip.spec.Tracker"/>
            </causes>
        </option>
    </causes>
</require>

<accept>
    <effect id="broadcast" specType="org.bip.spec.Tracker"/>
    <causes>
        <port id="speak" specType="org.bip.spec.Peer"/>
        <port id="listen" specType="org.bip.spec.Peer"/>
        <port id="broadcast" specType="org.bip.spec.Tracker"/>
    </causes>
</accept>
```

2.1.3 Data specification

JavaBIP components exchange data and make decisions concerning the components they want to interact with based on the data they receive. Data wires specify data that can be exchanged between components, by connecting the input data with the output data provided by other components. An example is provided in Figure 3.3.4.

2.2 JavaBIP module

A module comprises the functional code and the behavior specification of the corresponding component, as well as a dedicated executor. The behavior specification contains the Final State Machine (or FSM) with calls to the API methods provided by the component. It is used by the executor to drive the interaction with the engine and the environment.

Each executor is associated to a unique behavior specification. At each execution cycle, an executor computes the set of transitions enabled in the current state of the component (both enforceable and spontaneous). A transition is enabled when it has no guard or when its guard evaluates to true. The executor then uses the protocol presented in Section 1.1.

FSM transitions can be of three types: enforceable, spontaneous and internal.

Enforceable transitions are controlled by the engine. The executor sends a list of enabled enforceable transitions to the BIP Engine and waits for a response, indicating the port to be executed. Upon receiving this response, the BIP Executor performs the corresponding transition.

Spontaneous transitions are used to take into account changes in the environment and, therefore, they are not announced to the engine but rather executed after detection of events in the environment of the component.

Finally, internal transitions allow behavior specifications to update its state based on internal information when enabled, they are executed immediately. Spontaneous and internal transitions cannot be used for synchronization with other components.

2.3 JavaBIP engine

The engine comprises the kernel and a set of coordinators.

2.3.1 Engine kernel

The kernel combines and solves the various constraints of the system. Its implementation is based on Binary Decision Diagrams (BDDs) [1], which are efficient data structures to store and manipulate Boolean formulas. The kernel applies the three-step protocol described in Section 1.1.

In particular, it receives from the coordinators constraints in the form of Boolean formulas and assembles them by taking their conjunction to find a solution. The solution is sent back to the coordinators which interpret it and notify the components accordingly. The imposed constraints can be of two types:

- *Permanent constraints*: that are received only once at initialization. They include information about the Behavior, Glue and Data-wires of the components. In Figure 2.1, permanent constraints are shown with arrows labelled permanent.
- *Temporary constraints*: that are received at each execution cycle. They include information about the enabled transitions of components. In Figure 2.1, temporary constraints are shown with arrows labelled temporary.

2.3.2 Coordinators

Coordinators manage the flow of information between components and the engine kernel. They receive different types of information and encode them as Boolean constraints using dedicated encoders. Two coordinators that produce different types of constraints were devveloped: the BIP coordinator and the Data coordinator.

The BIP coordinator manages the information about the behavior, glue and current state of the components. It encompasses three dedicated encoders (cf. Figure 2.1): the Behavior encoder, Glue encoder and Current State encoder. The Boolean constraints encoding component behavior and glue are permanent, hence only computed by the Behavior and Glue encoders, respectively once at initialization. The Boolean constraints

[1] We have used the JavaBDD package, available at `http://javabdd.sourceforge.net`

encoding the current states of components are temporary, hence recomputed at each execution cycle.

The Data coordinator, relying on one encoder (Data encoder), is used on top of the BIP coordinator. It encodes as permanent constraints the information about data-wires, which connect input and output data provided by the components. At each execution cycle, the Data coordinator produces temporary constraints imposed on component interaction by the guards associated to component inputs. These temporary constraints disable the interactions involving data transfer, where the proposed output data values do not satisfy the guards associated to the corresponding input data. To this end, each guard that requires input data is evaluated on all data values, proposed along the data-wires attached to the corresponding port.

As shown in Figure 2.1, the coordinators form a chain. Depending on the needs of the application, different coordinators can be used. For instance, if there is data transfer, the Data coordinator must be used on top of the Glue coordinator. Otherwise, the use of solely the Glue coordinator is sufficient. Other coordinators can be easily added to manage other types of constraints - the implementation of the coordinator stack renders the architecture extensible.

2.3.3 Encoders

In order to transform the temporary and permanent constraints given by the component, we need the encoders to translate them so that the JavaBIP engine can understand them. We have four encoders for this purpose:

1. *Behaviour*: the behaviour encoder parses the annotated java classes such as in Figure 2.1.1 into a BDD;

2. *Glue*: the glue encoder parses the XML file as shown in Figure 2.1.2 into a BDD;

3. *Current state*: the current state encoder parses the temporary contraints (current state and disabled ports) into a BDD;

4. *Data*: the data encoder parses the XML as shown in Figure 3.3.4.

Chapter 3

Defining dynamicity in JavaBIP

We can now focus on dynamicity in JavaBIP, what it means, what are its challenges with the current implementation. We will also talk about tools we need to achieve dynamicity and the repercussions of dynamicity on the underlying structure of the core engine.

3.1 Dynamicity

There exists different types of dynamicity that can be implemented in BIP. For instance, Dy-BIP [8] is an extension of the BIP component framework that allows dynamic interactions among components that can evolve at run-time. Another type of dynamicity was implemented in a BIP adaptation to functional programming languages [9] where registration of components can happen on the fly if the ports of the newly registered components comply with a set of statically predefined port types.

In this thesis, we have worked on a similar type of dynamicity as the one implemented in the latter approach that involves dynamic registration of components at run-time. Additionally our approach allows deregistering components at runtime. In the current static implementation, whenever we want to add or remove a component, we need to shutdown the engine, update the system structure and relaunch an engine with those new components.

3.1.1 Implementation challenges

This kind of dynamicity adds a lot of flexibility to the existing JavaBIP framework but it comes with two main challenges.

Starting the engine For instance, in the Camel route example discussed in the introduction, it would be harmful to add routes without a monitor to manage the resources. And in more complex systems, it could be even more difficult to know when the engine has enough components to be started. So we need a mechanism to start the engine at the appropriate moment so this kind of error can not happen.

Removing components The second challenge consists in giving the ability to developers to dynamically remove components as well, we need a safety mechanism to not try to find solutions if there can not be any. For example, with the Camel routes system, we could have only a monitor and a route and decide to remove the route. Then we need to be able to support the case where we do not have enough components after having started the engine.

3.1.2 Dynamicity mechanisms

As discussed in previous sections, we have two main features in this extension of JavaBIP: adding and removing components on-the-fly, while the system is running.

- Adding a new component makes it part of the system and other components that were added beforehand will be able to interact with it. It is just as if the component was added prior to the engine starting;
- Removing a component completely removes it from the system. The system will not have any remaining references to the component and it would be as if it was never added to the system in the first place except for the change of state in the other components;
- There is another feature that is going to be available to the programmer. This feature is pausing a component. By pausing a component, it will stay in every data structure in the engine but none of the components will be able to interact with it. This is especially useful as a step towards *incrementality* because the component can pause itself if it sees that it will not have any transitions that the engine needs to handle. A programmer could also decide to pause a component to check its state. A component gets fully back in the system by informing the engine of its state and then can interact with the other components again.

3.2 Validity

In an effort to help the programmer, the responsibility of starting the system at the opportune moment now falls on the engine. To achieve this, we had to think about how do we know that we can start the engine and came up with the idea of the validity of a system.

3.2.1 Definition of system validity

First, we define the validity of a system as:

A system is valid if there are enough registered components such that there is at least one enabled interaction among them.

3.2.2 Determining system validity

Let $\mathcal{T}$ be the set of component types described in the glue. We consider the Require macros of the form p `Requires` a_1 `;` $\ldots$ `;` a_n in the glue definition of the system. Recall that an interaction a_i is a non-empty subset of enforceable ports. We denote $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T} \times \mathbb{N}$ the set of triples (t_1, t_2, n), such that there exists a macro t_1 `Requires` $t_2 \ldots t_2$ with the component type t_2 appearing n times. In order to determine the validity of a system, we consider the following graph $G = (\mathcal{T}, E)$, with component types as vertices and directed labeled edges of the form (t_1, t_2, n, c), where $(t_1, t_2, n) \in \mathcal{R}$ and $c \in \mathbb{Z}$ is a counter associated to the edge.

For OR clauses, we assign a color for each possible solution since only one of them needs to be satisfied. We will now explain what happens when we add or remove a component from the graph.

Adding a component

When adding a component of type t_1 to the graph, we take $E' \subseteq \mathcal{T} \times \mathcal{T} \times \mathbb{N}$ the set of edges, where $(t_2, t_1, n) \in E'$ means that there exists a macro "t_1 Requires t_2" with t_2 appearing n times and decrement the counter $c \in \mathbb{Z}$ of each edge in E'.

Removing a component

To remove a component of type t, we update the graph similarly as when adding a component, except that instead of decrementing the counters, we increment them.

Determining if the system is valid

We say an edge with counter $c \in \mathbb{Z}$ is valid if and only if $c \leq 0$.
Let $k \in \mathbb{N},\ i \in [0, k),\ C_i$ a conjunction of component types (e.g. $t_1t_1t_2t_3$ where $t_1,\ t_2,\ t_3 \in \mathcal{T}$). We say macro "$t$ Requires $C_1; C_2; \ldots; C_k$" is valid if at least one clause C_i is satisfied meaning that we have the required number of instances of each type present in the conjunction in the system. In other words, it means that the edges $(t,\ t',\ n)$ where $t' \in C_i$ all have a counter $c \leq 0$. And additionally, the nodes $t' \in C_i$ are satisfied.

Now, to determine if the system is valid using the graph when adding a component of type t, all we have to do is increment the counters and then check that all dependencies of the vertex t are satisfied. What this means is that we need to find at least one clause that is satisfied and then check that this clause has all its types satisfied.

When removing a component of type t for instance, we have to check every type that is connected to node t to see if the system became invalid. We do not want to check every type in the system because it is possible (see Figure 3.7) that the graph is not only one connected component.

So when adding/removing a component, we only need to verify that the connected component the node t is a part of is still valid or became valid and take the union of the boolean validity labels of all connected components of the system.

3.2.3 Validity proofs

Proposition 3.2.1. *If a system is valid and we add a new component, the system remains valid.*

Proof. Let $t \in \mathcal{T}$ be the type of the component we want wo add to the graph. We have three cases depending on the requirements of the component:

1. `t Requires -`: If the the component requires nothing, then whether the system was valid before or not, we still have a valid system with this component alone, hence the system is valid.

2. `t Requires` C: Let C be a clause, i.e. a conjunction of component types. We have again multiple cases:

   - Not all the edges of t are satisfied: As we know, when adding a component, we decrement the counter linked to each edge. If not all the edges are satisfied, it means that $\exists$ e $\in \mathcal{T} \times \mathcal{T} \times \mathbb{Z}$ such that $e = (t, t', i)$ where i > 0 for some $t' \in \mathcal{T}$. It implies that the new component does not have all its requirements satisfied and thus can not be part of a valid system. However, decrementing the counters means that we can not make a counter that was already negative or zero and make it strictly greater than zero. Hence it will not influence another valid system somewhere else in the graph.
   - all edges are satisfied: If all edges are satisfied for t, then in order to know whether the new component will be part of a new valid system, we need to check its requirements. If all are satisfied then we have a new valid system but if not, then we still have a valid system elsewhere and the same argument remains: when in $\mathbb{Z}$ decrementing an integer already less than or equal to 0 can not make it positive again. Hence the system remains valid.

3. `t Requires` C_1 `;` C_2 This case is similar to the previous one. We simply have more options to have a new valid system with the new component. The same argument can be made with clause C_1 as previously and if we do not have a valid system then, we can actually make the same again with C_2 and even if we still do not have a valid system, the fact that we decrement negative or zero integers in $\mathbb{Z}$ implies that they can not be positive again. And thus the system remains valid from the one we had in the assumption.

Proposition 3.2.2. *If a system is invalid and we remove a component, the system remains invalid*

Proof. In the previous proof, we saw that the determining factor in a system remaining valid after addition of a new component is that we decrement negative or zero integers in $\mathbb{Z}$. We can make the reverse argument here.

If the system is invalid, it means that enough edges have a strictly positive counter so that we can not have any interaction between components. Removing a component

of type t means that we are going to increment the counters of the edges linked to the node t. And here we notice that incrementing a strictly positive integer in $\mathbb{Z}$ returns a strictly positive integer so the edges that were making the system invalid before can not be satisfied after removal of said component. Hence, if the system was invalid before removal of the component of type t, it will still be invalid after.

3.2.4 Examples of validity graphs

Example 3.2.3. Camel routes: A Camel route transfers data among a number of data sources. 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. The Camel API provides the methods `resumeRoute` and `suspendRoute` to control the activation of a route. For simplicity, we assume here that the memory used by an active route is known, whereas the memory used by a suspended route is negligible.

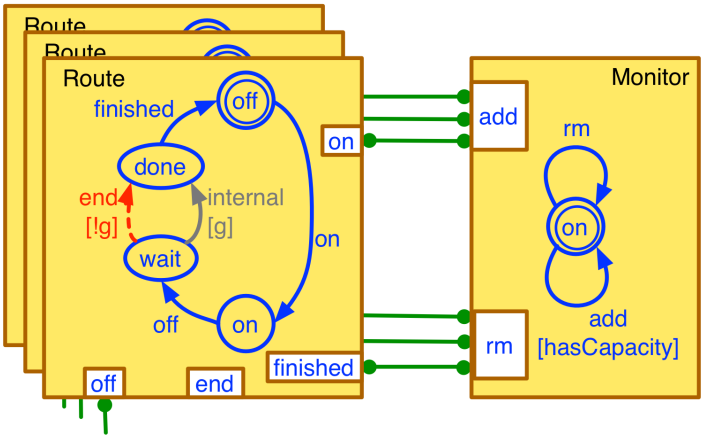


Figure 3.1: System graph for Camel routes

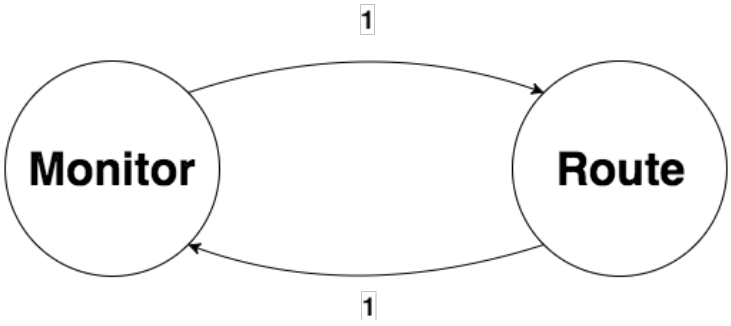


Figure 3.2: Validity graph for the Camel routes system

We use the glue in Figure 7 to get the graph in Figure 3.2. To begin with, we create

one node per component type described in the XML file: Monitor and Route. Then, we look at the require clauses but we can omit the ports because they are not of interest. If we simplify those clauses and remove redundancies, we have only two requirements remaining: `Route Requires Monitor` and `Monitor Requires Route`. This is how we then deduce both arrows labeled 1.

Example 3.2.4. Trackers and Peers: The following example presented in considers a simplified wireless audio protocol for reliable multicast communication. There are two component types: Tracker and Peer. The protocol allows an arbitrary number of peers to communicate along an arbitrary number of wireless communication channels. Each channel is managed by a unique tracker.

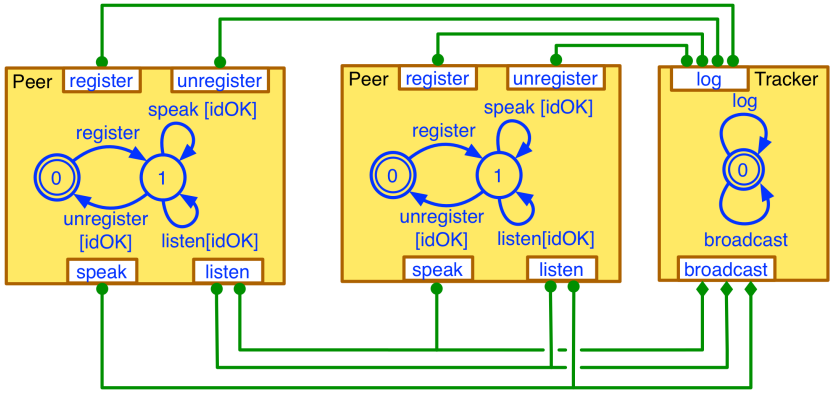


Figure 3.3: System graph for Trackers and Peers

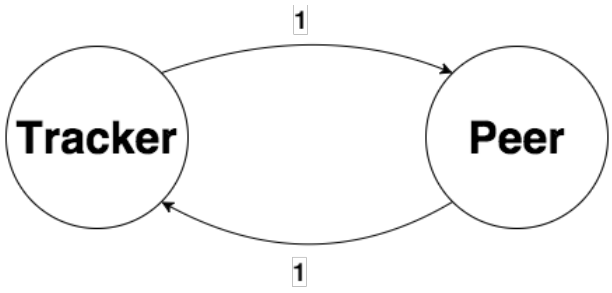


Figure 3.4: Validity graph for the trackers and peers system

Same as before, the minimal requirements are fairly simple (see Figure 3.4). If we simplify and decompose the glue as in the previous example, we see that we are left with `Tracker Requires Peer` and `Peer Requires Tracker`, hence the previous graph.

Example 3.2.5. Publishers and Subscribers

In this case, we see in Figure 3.6 that we are missing a few component types from the system such as *Topic* or *TopicManager*. This is a consequence of them not having any enforceable transitions and thus no need for the engine to manage the components

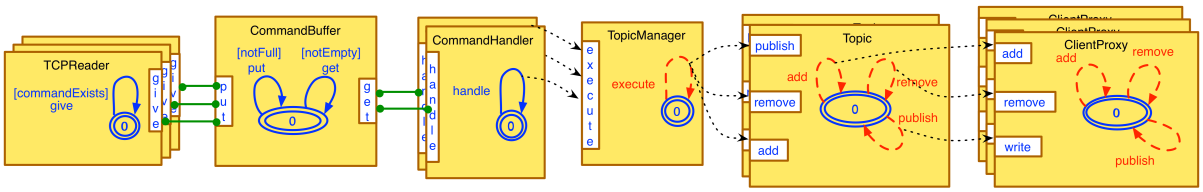


Figure 3.5: System graph for Publishers Subscribers

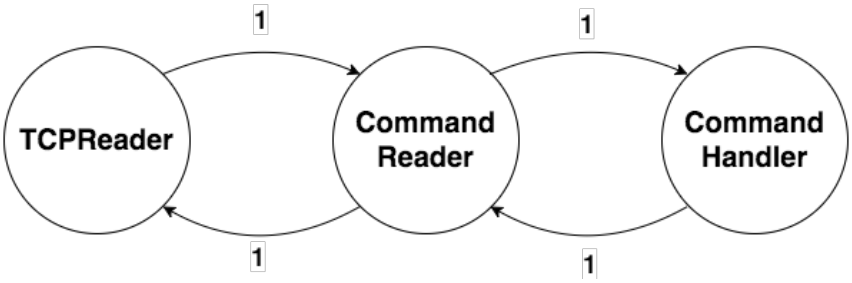


Figure 3.6: Validity graph for the publishers and subscribers system

of those types. Hence they are not in the glue and not taken into account by the graph. But if we simplify and decompose as in the previous examples, we are left with:

- `TCPReader Requires CommandReader`
- `CommandReader Requires TCPReader`
- `CommandReader Requires CommandHandler`
- `CommandHandler Requires CommandReader`

Hence the graph in Figure 3.6

Example 3.2.6. Example system

With this graph, we have a much more different system with more complex dependencies (see Figure 3.7). We have an OR clause which makes it possible to have multiple possible solutions (represented with colored arrows on the graph).

This example is more complex to decompose but we have the result:

- `A Requires B B`
- `B Requires A A A C ; C C`
- `C Requires -`
- `D Requires E`
- `E Requires -`

The OR clause representing multiple solutions is shown as different colors (green and red) in Figure 3.7.

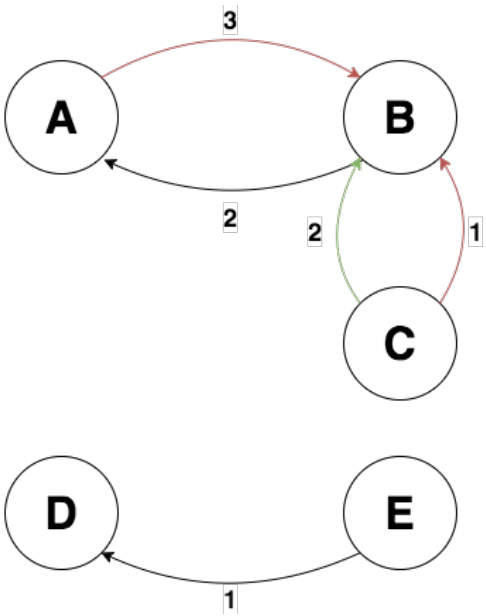


Figure 3.7: Validity graph for the example system

3.3 Boolean encoding using BDDs

In this section, we present the modifications that we have done on the various encoders used in the JavaBIP coordinators, in order to allow them to handle dynamicity.

Let S_i be the set of states of component i and $P_i(x)$ be the state of ports of state x for component i.

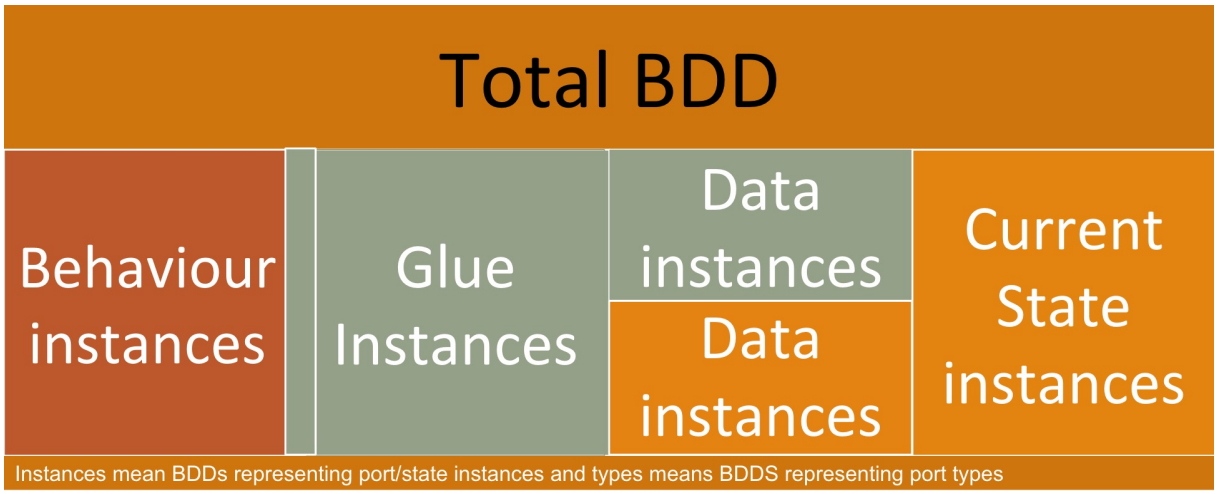


Figure 3.8: BDDs in the JavaBIP system

Figure 3.8 represents each BDD which are then conjuncted to form the total BDD.

Dark orange represents permanent constratints which are given only once at initialization. Grey represents information that needs recomputation each time we register a new component and dark yellow represents data that is sent at each cycle. Behaviour BDD is mostly given at initialization (i.e. before the engine starts) but each time a component registers, we still need to conjunct its behaviour with the total behaviour BDD. The glue BDD however needs to be recomputed each time a component registers, same as for some data constraints. Other data constraints and current state constraints though are sent at each cycle.

3.3.1 Behaviour encoding

First of all, the Boolean encoding of the component behaviour is computed as:

$$behaviourBDD_i = \bigwedge_{s \in S_i} \left(s \quad \wedge \quad \bigwedge_{p \in P_i(s)} p \quad \wedge \quad \bigwedge_{s' \in S_i - s} \left(\overline{s'} \wedge \bigwedge_{p' \in P_i(s')} \overline{p'} \right) \right) \tag{3.1}$$

Here is an example for a route of the Camel routes system (see Figure 3.1) where x_s is the state named x and x_p is the port named x:

$$\begin{aligned} behaviourBDD = &\left(off_s \wedge on_p \wedge \left(\left(\overline{on_s} \wedge \overline{off_p}\right) \wedge \overline{wait_s} \wedge \left(\overline{done_s} \wedge \overline{finished_p}\right)\right)\right) \wedge \\ &\left(on_s \wedge off_p \wedge \left(\left(\overline{off_s} \wedge \overline{on_p}\right) \wedge \overline{wait_s} \wedge \left(\overline{done_s} \wedge \overline{finished_p}\right)\right)\right) \wedge \\ &\left(wait_s \wedge \left(\left(\overline{off_s} \wedge \overline{on_p}\right) \wedge \left(\overline{on_s} \wedge \overline{off_p}\right) \wedge \left(\overline{done_s} \wedge \overline{finished_p}\right)\right)\right) \wedge \\ &\left(done_s \wedge finished_p \wedge \left(\left(\overline{off_s} \wedge \overline{on_p}\right) \wedge \left(\overline{on_s} \wedge \overline{off_p}\right) \wedge \overline{wait_s}\right)\right) \end{aligned} \tag{3.2}$$

And then, we take the conjunction of all of them to compute the total behaviour BDD of the system for $n \in \mathbb{N}$ components:

$$totalBehaviourBDD_n = \bigwedge_{i=0}^{n} behaviourBDD_i \tag{3.3}$$

As we can see, each cmoponent's behaviour BDD depends solely on the component itself. Since we take the conjunction of all those BDDs to get the total behaviour BDD, we can apply the same idea to adding components on-the-fly:

$$totalBehaviourBDD_{n+1} = totalBehaviourBDD_n \wedge behaviourBDD_{n+1} \tag{3.4}$$

3.3.2 Current state encoding

This temporary BDD is recomputed at each cycle when the component i gives informations such as its current state $cs \in S_i$ and the list of disabled ports for this state $dps \subseteq P_i(cs)$. After receiving the information, the current state BDD is computed as flollows:

$$currentStateBDD_i = cs \quad \wedge \left(\bigwedge_{s \in S_i - cs} \overline{s} \right) \quad \wedge \left(\bigwedge_{p \in dps} \overline{p} \right) \tag{3.5}$$

Here is an example for a route of the Camel routes system (see Figure 3.1) where x_s is the state named x and x_p is the port named x with the route at state *on*:

$$currentStateBDD = on_s \wedge \left(\overline{off_s} \wedge \overline{wait_s} \wedge done_s \right) \tag{3.6}$$

(We have no disabled ports for state *on* because there are no guards)

Since this is recomputed at each cycle, we do not have to make any changes with this BDD because the total current state BDD is computed as:

$$totalCurrentStateBDD_n = \bigwedge_{i=0}^{n} currentStateBDD_i \tag{3.7}$$

with n changing for the cycle after addition.

3.3.3 Glue encoding

This BDD is computed though the glue specification that is given at initialization. The glue BDD depends on the instances that we have in the system which makes it a BDD that needs to be modified in some way at addition or removal of a component.

In the current implementation, *Requires* macros strictly enforce the cardinality of the instances of a type. So for the following macro `T.x Requires U.y` with instances t_0 and u_0 of types T and U respectively, we would have to following BDD:

$$t_0.x \Rightarrow u_0.y \tag{3.8}$$

However for two instances of type U, u_0 and u_1, we would then have:

$$t_0.x \Rightarrow (u_0.y \wedge \overline{u_1.y}) \vee (\overline{u_0.y} \wedge u_1.y) \tag{3.9}$$

which is equivalent to:

$$t_0.x \Rightarrow u_0.y \oplus u_1.y \tag{3.10}$$

So even with only a simple macro, adding or removing a component means changing the BDD and not just adding a new BDD to it which is something cumbersome. The fact that the glue BDD depends on the instances in the system is a limitation. If it did not, we would not have to change anything and the glue BDD would be permanent and never need a change.

3.3.4 Data encoding

The Data BDD is constructed differently. First, we need to create new data variables (or d-vars) for each instance of a data wire. So for example, if we have a data wire such as:

```
<data>
    <wire>
        <from id="trackerId" specType="org.bip.spec.Tracker"/>
        <to id="trackerId" specType="org.bip.spec.Peer"/>
    </wire>
</data>
```

With two instances of $org.bip.spec.Peer$ p_0 and p_1 and one of $org.bip.spec.Tracker$ t_0, we would have the following BDDs (see Figure 3.3 for a reminder on the Trackers and Peers system):

$$\begin{aligned} d_0 &\Rightarrow p_0.register \wedge t_0.trackerId \\ d_1 &\Rightarrow p_1.register \wedge t_0.trackerId \\ p_0.register &\Rightarrow d_0 \\ p_1.register &\Rightarrow d_1 \end{aligned} \tag{3.11}$$

Let ND the set of ports needing data, PD the set of ports providing data, $DW \subseteq T^2$ the abstract datawire types and DV the instances of the datawires (d-vars) and $DV(p)$ where p is a port and U is the type of the component with port p, the d-var of a datawire as $(U, \_)$ or $(\_, U)$. In general, the equations are:

$$\begin{aligned} \forall i \in \mathbb{N} \mid p_j \in ND,\ p_k \in PD,\ \exists d_i \in DV(p_j) \cap DV(p_k), \\ d_i &\Rightarrow p_j \wedge p_k \\ p_j &\Rightarrow \bigcup_{d \in DV(p_j)} d \end{aligned} \tag{3.12}$$

As you can see, if we want to add a new instance of any type using data in the system, we need to be careful. We need to check for every datawire if the new component needs the data or provides it. For either, we will need to create 1 d-var per possibility and then create each $d_i \Rightarrow p_j \wedge p_k$ BDDs. However. We cannot simply add new BDDs for the implications from ports to d-var because adding a new component like a tracker in our example would change these BDDs:

$$\begin{aligned} d_0 &\Rightarrow p_0.register \wedge t_0.trackerId \\ d_1 &\Rightarrow p_1.register \wedge t_0.trackerId \\ p_0.register &\Rightarrow d_0 \\ p_1.register &\Rightarrow d_1 \end{aligned} \tag{3.13}$$

into these:

$$\begin{aligned} d_0 &\Rightarrow p_0.register \wedge t_0.trackerId \\ d_1 &\Rightarrow p_1.register \wedge t_0.trackerId \\ d_2 &\Rightarrow p_0.register \wedge t_1.trackerId \\ d_3 &\Rightarrow p_1.register \wedge t_1.trackerId \\ p_0.register &\Rightarrow d_0 \vee d_2 \\ p_1.register &\Rightarrow d_1 \vee d_3 \end{aligned} \tag{3.14}$$

And we cannot simply slightly modify those BDDs, we need to recompute the second part but only that part.

Chapter 4

Implementation

In this chapter, we will explain in details the implementation of the dynamicity in JavaBIP.

For this chapter, the BIPEngine will be mentioned as engine, not to be confused with the core engine which computes the possible solutions depending on the behaviour, glue and data constraints.

To begin with, we will cover every modified function on the engine's interface and changes made to the core engine and encoders to make dynamicity work with the engine. This includes added and deleted functions to the engine's interface and added functionality to the core engine as well as existing functions that had to be modified to work with the static and the dynamic scenarios that the engine could encouter.

To conclude, we will present a new data structure that greatly helps modularize the work within the engine.

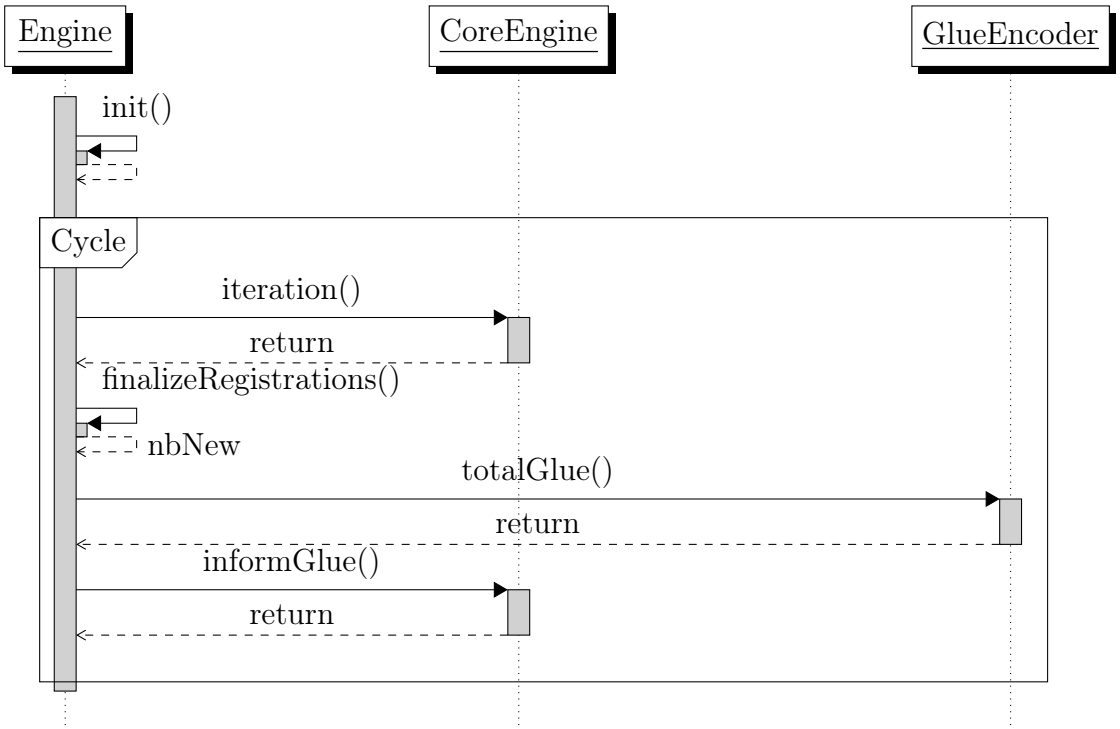


Figure 4.1: Execution cycle of the engine

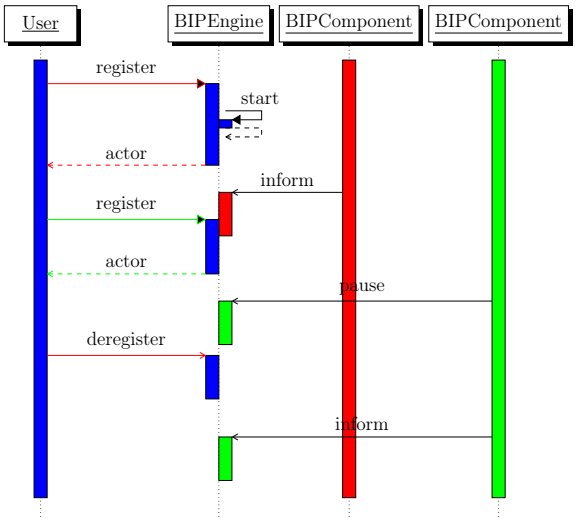
4.1 ComponentPool

The *ComponentPool* is an object that is used as an interface to the graph described in section 3.2.2. It returns a boolean value whenever a component is added or removed reflecting the validity of the system.

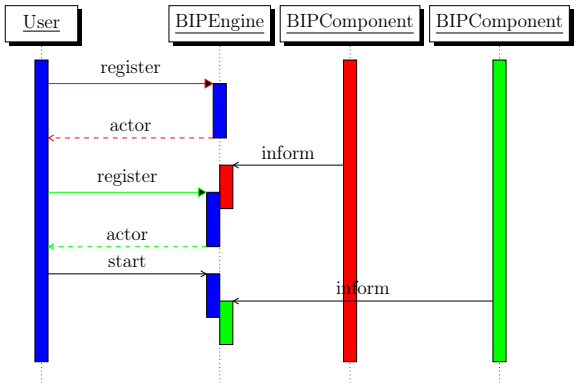
4.2 The BIPEngine interface

In the static implementation of JavaBIP, the engine's interface constituted of the following functions:

- `register`: Registers a component with the system so it can be a part of the interactions of the system;
- `inform`: Sends information from the executor to the engine saying that a *BIPComponent* is at state S and has a *List<Port>* that are disabled for S;
- `informSpecific`: Sends information to the engine to specify that for a component and a port, we have a list of disabled ports per other component with which we cannot interact (used mostly in the data coordinator);
- `specifyGlue`: Sends the glue to the engine;



(a) Dynamic workflow



(b) Static workflow

Figure 4.2: Differences in the workflows between dynamic and static implementations

- `start`: Starts the engine thread but waits at the initialization cycle that the *execute* function has been called;
- `execute`: Launches the initialization cycle of the engine and sets which coordinator will send messages to the *BIPComponent* when transitions have been decided (i.e. *InteractionExecutor*);
- `stop`: Stops the engine;
- `initialize`: Initializes the context for typed actor.

Some of those functions were slightly changed such as `initialize` and other left untouched like `specifyGlue`. Others, however, had to be substantially modified such as `register` and `inform`.

4.2.1 Starting and executing the engine

Starting the engine stays the same as in the static implementation, although now that the engine can start itself, we do not need this functions to be in the engine's interface anymore.
A problem with this feature is that when registering a component, we cannot start the engine from the *BIPCoordinator* because if we have data, the *DataCoordinator* has not finished registering the component yet. Hence, the decision of starting the engine must belong to the topmost coordinator through a callback the *BIPCoordinator* would set.

We have a more private interface called *EngineStarter* seen only by the coordinators. The *EngineStarter* is selected at creation of the engine depending on which coordinator

is the topmost one. So at the end of registration, the *BIPCoordinator* sets the callback to start the engine to the *EngineStarter* of the engine. Thus, the correct coordinator will start the engine when we are sure the component is completely registered.

4.2.2 Initializing the engine

At initialization, we set the *EngineStarter* of the engine and the *DataInformer* so we know which coordinator it is (if any) before letting any component register.

4.2.3 Registering a new component

The previous registration process was taking into account that the engine was not running yet so not finding solutions to the constraints (i.e. using the BDDs). This time around, we have to be careful of multiple scenarios in which we could be and that we do not leave the core engine in an inconsistent state by modifying the BDDs during an iteration.

After creating the *BIPComponent* for the instance we want to register, we can update several data structures such as adding the component to the set of registered components and map its behaviour to the itself. Then, we create the BDD nodes for this component's behaviour since it does not impact the behaviour BDD and if we are not currently running, we can update the number of components and inform the newly created behaviour BDD to the core engine and update corresponding data structures in the behaviour encoder.

However, we cannot simply return the component now. We have to be careful to what state the engine is in. As stated before, we inform the core engine of the behaviour of the component only if the engine is *not* currently running.
After adding the component to the *ComponentPool*, this leaves us with several new cases:

1. The system is not valid and the engine is not running:
   This means that the engine is not ready to be started yet. We can follow the static implementation and return the component.

2. The system is valid but the engine is not running:
   This means that with this new addition, we have a valid system and we can finally start the engine. So we tell the *EngineStarter* that when it is done with the registration, it can start the engine.

3. The component makes the system valid but the engine is running:
   This case can be deceiving as it could mean that we encountered an inconsistency because the engine should not be running if we do not have a valid system. However, with the additions of `deregister` and `pause` to the *BIPEngine* API, we can end up in a state where we do not have enough components anymore to run.
   In this event, we pause the engine and reaching this case means that we can finish the registration of the component (computing all relevant BDDs and informing the

engine and finally recomputing the total BDDs) and wake up the engine because we can continue the execution.

4. The system is valid and the engine is running:
   This means that we are currently running and have to be very careful with the information we give the core engine. This is called a registration on-the-fly.
   Here, after the engine finishes its iteration, we prevent it from beginning a new one before we are done registering this comoponent. So we compute the behaviour BDD and inform the core engine and tell the engine that we have this new component so it can recompute the glue BDD and the total behaviour BDD.
   So for the next iteration, the component will be part of the system.

This concludes the presentation of the new `register` function. It needs to be aware of the state of the engine and the state of the system and the impact adding the new component would make when added to it.

4.2.4 Informing the engine

Whenever a component informs, given its current state and the disabled ports for this state, it will ask the relevant encoder to transform those constraints into a temporary BDD and inform the engine so it can find a proper solution given each component's situation.

One important thing to know is that a component will inform as soon as it is created. So if the engine is running and computing new solutions, informing it of the current state of a new component not even in the glue BDD or the total behaviour BDD could have two consequences: either we have an inconsistent solution with the actual system (which the new component might not yet be a part of) or simply crash the engine.

We obviously cannot let either happen. The problem actually occurs whenever we have a registration on-the-fly. If we look closely, a component informing when the engine is either not running or paused is tolerable. So the only problem is when the engine is doing some actual work.

In this case, let us remind that we are aware of which components are being registered on-the-fly so the engine can recompute the total behaviour BDD and the glue BDD. We can use this knowledge to block the component from informing the core engine before its registration is finalized. Using this, we avoid both cases and let the component inform when the engine is ready to have this information.

Another addition to this implementation is the handling of the case when an unregistered component tries to inform because it might not have been disposed of yet. Instead of it being a fatal case, we can just ignore this information.

4.2.5 Stopping the engine

Before the dynamic implementation, we would simply call `thread.stop()` on the thread running the engine but this method is deprecated and is not considered good style.
Instead, in the new implementation, we call `thread.interrupt()` which will signal the thread that its execution has to be stopped at some point (e.g. during a `thread.wait()` or in our case `semaphore.acquire()`) and continue the cycles as long as the thread was not interrupted.
So within the stop method, we can call `thread.join()` which will wait for the thread to reach one of those points and we can end the execution in a cleaner way.

Chapter 5

Performance

In this chapter, we are going to show a few numbers to see if JavaBIP did not slow down too much after adding dynamicity. We chose the example system described in Figure 3.7 for testing without data and the Trackers and Peers system described in Figure 3.3 for testing with data.

5.1 Registration

In the Figures 5.1 and 5.2, color blue represents a registration before starting the engine, red is when registration makes the engine start or yellow for after the engine has started and green for the cycles.

5.1.1 Registration before engine start

Registration before the engine start is represented by the blue chart on both Figures 5.1 and 5.2.

Without data

As we can see in Figure 5.1, registration time before the engine has to be started is around the 35 to 45 ms mark. The median of this sample is 37ms and the average 44.6875ms.

With data

With data, the time is slightly more important but is still reasonable. We have a median of 43ms and an average of 53.875ms.

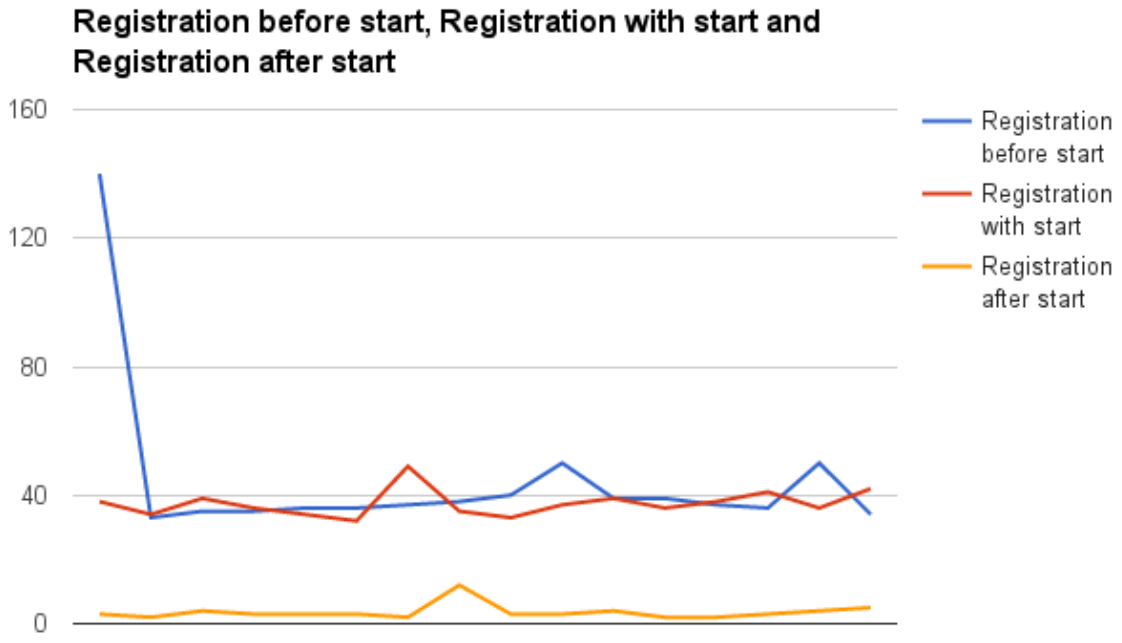


Figure 5.1: Chart showing example system time in function of the iteration number

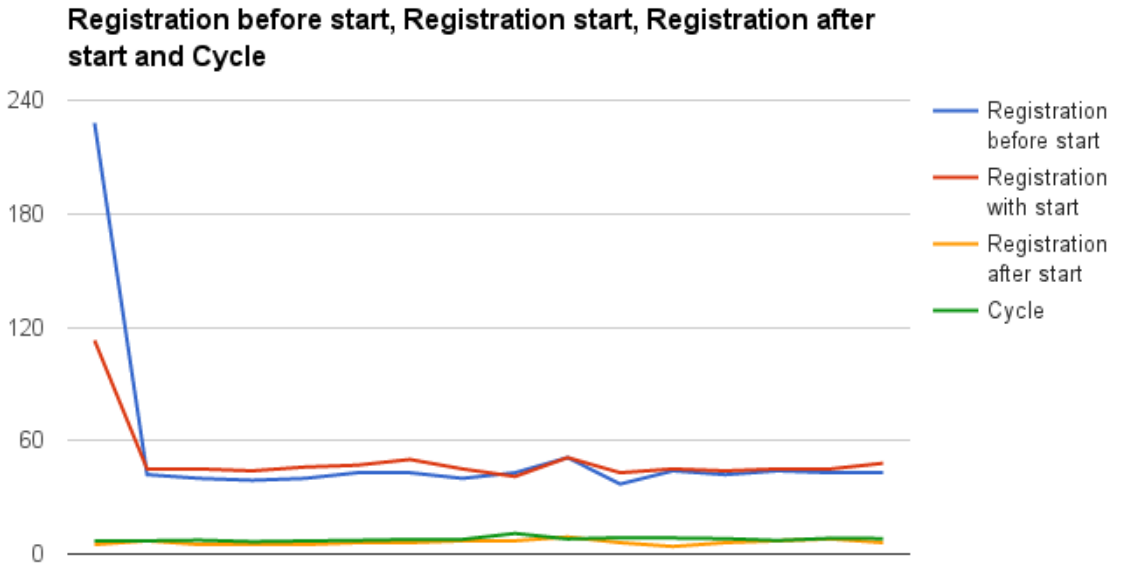


Figure 5.2: Chart showing trackers and peers time in function of the iteration number

5.1.2 Registration with engine start

This part refers to when registering a particular component makes the system valid and we have to start the engine. These results are quite similar to the previous category. They are represented by the red chart in both Figures 5.1 and 5.2.

Without data

When we need to start the engine, the median is 36.5ms and the average is 37.4375ms which is very close to before the engine start. This is mostly because starting the engine is done through a callback and so there is no extra work that has to be done by the component to start the engine.

With data

For the same reason as before, the results are close to before starting the engine with a median of 45ms and an average of 49.8125ms.

5.1.3 Registration on-the-fly

The results for the registration on-the-fly is represented by the yellow chart on Figures 5.1 and 5.2. Those results are very low compared to both previous categories.

Without data

In the example system, the median is 3ms and the average is 3.625ms.

With data

In the Trackers and Peers system, the median is 6ms and the average is 6.1875ms.

5.2 Cycles

This category measures the time spent in cycles by the engine. We measured only cycles lasting more than 5ms since a lot of cycles were lasting 0 to 4 ms but doing nothing. Without data, the median was 5.68ms and with data, it was 7.71ms which seems appropriate since we have one more coordinator and more BDDs.

Chapter 6

Future work

Even if being able to add and remove instances in a system is a great feature to have in JavaBIP, we feel that there is room for a few other features related to dynamicity. Below is a non-exhaustive list of ideas that can be inplemented in the future:

- Registering batches of components:
  In the present implementation, the glue BDD is recomputed each time a component is added. If a programmer wants to register a lot of components simultaneously , the engine performance can be optimised by skiping the unnecessary BDD recomputation. We could have a mechanism that lets us register a whole batch of components and at the end of this registration, let the engine do its job and finalize all registrations at the same time so we recompute the glue BDD only once.

- Registering types:
  This feature pushes further the concept of dynamicity in JavaBIP. Instead of just adding instances of types that are described in the glue, we would let the user add new rules to the glue and maybe even modify the existing ones or removing others. Then, the programmer could just register the new instance of this new type in the system as before and it would automatically add the new type to the glue BDD. Modifying the glue could easily be done with the *GlueBuilder* that is available to describe the glue before running the engine.

- A new kind of glue:
  As seen in Section 3.3.3, the glue BDD depends on each instance in the system which makes it a semi-permanent BDD. It does not need to change every cycle but it does need to change everytime we add a component. What we propose here is to make the glue BDD instances independent, and have an entire BDD for instances. We would need to create new BDD nodes for types and make the glue BDD with them. For example, if we have the following macro:

  ```
  T.x Requires U.y U.y ; U.y V.z
  ```

  with instances t_0, u_0, u_1, v_0, we would have the following BDD:

  $$t_0.x \Rightarrow (u_0.y \wedge u_1.y) \vee ((u_0.y \oplus u_1.y) \wedge v_0.z) \tag{6.1}$$

but with this new glue, we would have:

$$\begin{aligned} T &\Rightarrow U_2 \vee (U_1 \wedge V) \\ T &\Rightarrow t_0 \\ U_2 &\Rightarrow u_0 \wedge u_1 \\ U_1 &\Rightarrow u_0 \oplus u_1 \end{aligned} \tag{6.2}$$

So if we add a new instance of type T, all we need to do is recompute the BDD $T \Rightarrow t_0$. Same for any instance of any other type.

This idea is not complete nor is verified for correctness but the concept of not having to recompute everything all the time seems like a worthy optimization and it makes more sense for the glue to be instances independent as well since it is not described as dependent by the programmer.

Chapter 7

Conclusion

In conclusion, static JavaBIP allowed to create large concurrent systems but by being static, adding and removing component meant shutting down the whole system. This lead to losing all the progress and the results of interactions prior to addition or removal of components.

However, dynamic JavaBIP addresses this limitation by allowing developers to add and remove those components on-the-fly. The new JavaBIP engine also relieves developers from the responsibility of starting the engine by having the *ComponentPool* supporting a graph that can tell the engine whenever there are enough components in the system for it to be valid. At that moment, we can automatically start the engine.

Something similar also happens when removing components. The engine can know when it does not have enough components anymore to find interactions and it can pause itself waiting for a new component to come along and make the system valid again.

The *ComponentPool* can be used in other features such as incrementality as well and dynamic JavaBIP can be extended with dynamic addition of types and addition of batches of components.

Appendix

Listing 7.1: Require Glue of the Camel Routes system

```
<require>
    <effect id="on" specType="Route"/>
    <causes>
        <option>
            <causes>
                <port id="add" specType="Monitor"/>
            </causes>
        </option>
    </causes>
</require>
<require>
    <effect id="finished" specType="Route"/>
    <causes>
        <option>
            <causes>
                <port id="rm" specType="Monitor"/>
            </causes>
        </option>
    </causes>
</require>
<require>
    <effect id="add" specType="Monitor"/>
    <causes>
        <option>
            <causes>
                <port id="on" specType="Route"/>
            </causes>
        </option>
    </causes>
</require>
<require>
    <effect id="rm" specType="Monitor"/>
    <causes>
        <option>
            <causes>
                <port id="finished" specType="Route"/>
            </causes>
        </option>
    </causes>
```

```
</require>
<require>
    <effect id="off" specType="Route"/>
    <causes>
        <option>
            <causes>
            </causes>
        </option>
    </causes>
</require>
```

Listing 7.2: Require Glue of the Trackers and Peers system

```
<require>
    <effect id="register" specType="org.bip.spec.Peer"/>
    <causes>
        <option>
            <causes>
                <port id="log" specType="org.bip.spec.Tracker"/>
            </causes>
        </option>
    </causes>
</require>
<require>
    <effect id="speak" specType="org.bip.spec.Peer"/>
    <causes>
        <option>
            <causes>
            </causes>
        </option>
    </causes>
</require>
<require>
    <effect id="listen" specType="org.bip.spec.Peer"/>
    <causes>
        <option>
            <causes>
            </causes>
        </option>
    </causes>
</require>
<require>
    <effect id="unregister" specType="org.bip.spec.Peer"/>
    <causes>
        <option>
            <causes>
                <port id="log" specType="org.bip.spec.Tracker"/>
            </causes>
        </option>
    </causes>
</require>
  <require>
    <effect id="broadcast" specType="org.bip.spec.Tracker"/>
```

```
41      <causes>
42          <option>
43              <causes>
44              </causes>
45          </option>
46      </causes>
47  </require>
48    <require>
49      <effect id="log" specType="org.bip.spec.Tracker"/>
50      <causes>
51          <option>
52              <causes>
53                  <port id="unregister" specType="org.bip.spec.Peer"/>
54              </causes>
55          </option>
56               <option>
57              <causes>
58                  <port id="register" specType="org.bip.spec.Peer"/>
59              </causes>
60          </option>
61      </causes>
62  </require>
```

Listing 7.3: Require Glue of the example system

```
1   <require>
2       <effect id="portA" specType="org.bip.spec.ExampleA" />
3       <causes>
4           <option>
5               <causes>
6                   <port id="portB" specType="org.bip.spec.ExampleB" />
7                   <port id="portB" specType="org.bip.spec.ExampleB" />
8               </causes>
9           </option>
10      </causes>
11  </require>
12  <require>
13      <effect id="portB" specType="org.bip.spec.ExampleB" />
14      <causes>
15          <option>
16              <causes>
17                  <port id="portA" specType="org.bip.spec.ExampleA" />
18                  <port id="portA" specType="org.bip.spec.ExampleA" />
19                  <port id="portA" specType="org.bip.spec.ExampleA" />
20                  <port id="portC" specType="org.bip.spec.ExampleC" />
21              </causes>
22          </option>
23          <option>
24              <causes>
25                  <port id="portC" specType="org.bip.spec.ExampleC" />
26                  <port id="portC" specType="org.bip.spec.ExampleC" />
27              </causes>
28          </option>
```

```
    </causes>
</require>
<require>
    <effect id="portC" specType="org.bip.spec.ExampleC" />
    <causes>
        <option>
            <causes>
            </causes>
        </option>
    </causes>
</require>
<require>
    <effect id="portD" specType="org.bip.spec.ExampleD" />
    <causes>
        <option>
            <causes>
                <port id="portE" specType="org.bip.spec.ExampleE" />
            </causes>
        </option>
    </causes>
</require>
<require>
    <effect id="portE" specType="org.bip.spec.ExampleE" />
    <causes>
        <option>
            <causes>
            </causes>
        </option>
    </causes>
</require>
```

Bibliography

[1] G. Apha, *Actors: a model of concurrent computation in distributed systems.* 1986.

[2] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE Software*, no. 28, pp. 41–48, 2011.

[3] S. Bliudze and J. Sifakis, "The algebra of connectors - structuring interaction in bip," *IEEE Transactions on Computers*, no. 57, pp. 1315–1330, 2008.

[4] S. Bliudze, J. Sifakis, M. Bozga, and M. Jaber, "Architecture internalisation in bip," *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, pp. 169–178, 2014.

[5] S. Bliudze, A. Mavridou, R. Szymanek, and A. Zolotukhina, "Coordination of software components with bip: application to osgi," in *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, pp. 25–30, ACM, 2014.

[6] S. Bliudze, A. Mavridou, R. Szymanek, and A. Zolotukhina, "For coordination, state component transitions," in *EclipseCon Europe 2013*, no. EPFL-TALK-196424, 2013.

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

[8] M. Bozga, M. Jaber, N. Maris, and J. Sifakis, "Modeling dynamic architectures using dy-bip," in *Proceedings of the 11th International Conference on Software Composition*, SC'12, (Berlin, Heidelberg), pp. 1–16, Springer-Verlag, 2012.

[9] R. Edelmann, "Behaviour-interaction-priority in functional programming languages: Formalisation and implementation of concurrency frameworks in haskell and scala," tech. rep., 2015.