SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2016; **00**:1–38
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Exogenous Coordination of Concurrent Software Components with JavaBIP

Simon Bliudze[1]*, Anastasia Mavridou[2], Radoslaw Szymanek[3] and Alina Zolotukhina[1]

[1]*Ecole polytechnique fédérale de Lausanne 1015, Lausanne, Switzerland*
[2]*Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37235, USA*
[3]*Crossing-Tech S.A., EPFL Innovation Park, 1015, Lausanne, Switzerland*

SUMMARY

A strong separation of concerns is necessary in order to make the design of domain-specific functional components independent from cross-cutting concerns, such as concurrent access to the shared resources of the execution platform. Native coordination mechanisms, such as locks and monitors, allow developers to address these issues. However, such solutions are not modular, they are complex to design, debug and maintain. We present the JavaBIP framework that allows developers to think on a higher level of abstraction and clearly separate the functional and coordination aspects of the system behavior. It implements the principles of the BIP component framework rooted in rigorous operational semantics. It allows the coordination of existing concurrent software components in an exogenous manner, relying exclusively on annotations, component APIs and external specification files. We introduce the annotation and specification syntax of JavaBIP and illustrate its use on realistic examples; present the architecture of our implementation, which is modular and easily extensible; provide and discuss performance evaluation results. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: BIP; JavaBIP; Component coordination; Concurrency; Modularity

*Correspondence to: EPFL IC IINFCOM LCA2, Station 14, 1015 Lausanne, Switzerland.
E-mail: Simon.Bliudze@epfl.ch

CONTENTS

1 Introduction **3**
1.1 The BIP Component Framework 4
1.2 Paper Contribution and Structure 4

2 JavaBIP Design Workflow **5**

3 JavaBIP by examples **6**
3.1 Camel routes 7
3.2 Publish-Subscribe server 10
3.3 Trackers and Peers example 12

4 Theoretical foundations **13**
4.1 Component model without data 13
4.1.1 Components and glue 13
4.1.2 Connectors 14
4.1.3 Boolean encoding of the connectors and macro notation 15
4.1.4 Macro notation based on component types 17
4.2 Extension of the model with data 19
4.2.1 Components with data 19
4.2.2 Composition 21

5 System specification **22**
5.1 Behavior specification 22
5.2 Glue specification 23
5.3 Data-wire specification 26

6 Implementation **26**
6.1 JavaBIP module 27
6.2 JavaBIP engine 28
6.2.1 Engine kernel 28
6.2.2 Glue coordinator 29
6.2.3 Data coordinator 30
6.3 Experimental evaluation 30

7 Related work **31**

8 Conclusion and Future Work **35**

DOI: 10.1002/spe

1. INTRODUCTION

When building large concurrent systems, one of the key difficulties lies in coordinating component behavior and, in particular, management of the access to shared resources of the execution platform. This is well illustrated by our motivating use-case, which consists in managing the memory usage by a set of Camel routes [1]. Camel routes are extensively utilized in Connectivity Factory™—the flagship product of Crossing-Tech S.A. 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.

In all mainstream programming languages, including Java and C++, basic coordination primitives are implemented as built-in features of the language [2, 3]. Different variations of locks, semaphores and monitors are used to express coordination constraints. However, these low-level primitives are mixed up with the functional code, forcing developers to keep both aspects simultaneously in mind—not only at design time, but also during debugging and maintenance. Since in concurrent environments it is practically infeasible to envision all possible execution scenarios, synchronization errors can result in race conditions and deadlocks.

Furthermore, an observable trend in software engineering is the increasing utilization of the declarative design techniques. 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. It is not always possible to instrument or even access the actual source code. Even if the code is generated explicitly, it is usually not desirable to modify it, since this can lead to a considerable increase of maintenance costs. This precludes the use of low-level primitives for the coordination of concurrent components that have to be reusable in different configurations and assemblies. Therefore, the problem of coordinating concurrent components calls for an exogenous solution that would allow developers to think on a higher level of abstraction, separating functional and coordination aspects of the system behavior.

To address this concurrency challenge, we have developed JavaBIP—a Java adaptation of the BIP (Behavior, Interaction, Priority) framework [4]—relying on the following observations. Domain specific components, such as Camel routes, have states (e.g., idle, working, suspended) that are known to component users with domain expertise. Furthermore, components always provide APIs that allow programs to invoke operations (e.g., suspend or resume) in order to change their state, or to be notified when the component changes the state spontaneously. Thus, component behavior can be represented by Finite State Machines (FSM). An FSM has a finite set of states and a finite set of transitions between these states. Transitions are associated with calls to API functions, which force a component to take an action, or with event notifications that allow reacting to external events coming from the environment. Since such states and transitions have intuitive meaning for developers, representing components as FSMs is an adequate level of abstraction for reasoning about their behavior.

To use JavaBIP, developers must provide—for the relevant components—the corresponding FSMs in the form of annotated Java classes. These classes are used to drive the interaction with the corresponding components through their provided APIs. The FSMs describe the protocol that must be respected to access a shared resource or use a service provided by a component, ensuring correct utilization of component interfaces.

For component coordination, JavaBIP provides two primitive mechanisms: 1) multi-party synchronization of component transitions and 2) asynchronous event notifications. The latter embodies the reactive programming paradigm. In particular, JavaBIP extends the Actor model [5], since event notifications can be used to emulate asynchronous messages. Providing the synchronization of component transitions as a primitive mechanism gives the developers a powerful and flexible tool to manage coordination.

DOI: 10.1002/spe

JavaBIP clearly separates system-wide coordination policies from the component behavior. Interaction constraints, defining the possible synchronizations among transitions of different components, are specified in XML configuration files independently from the design of individual components. For coordination scenarios that require global state information, dedicated *monitor* components can be added. This allows one to centralize all the information related to coordination in one single location, instead of distributing it across the components. Coordination is applied in an exogenous manner, relying totally on component APIs. Furthermore, JavaBIP components do not carry coordination logic that relies on the characteristics of any specific execution environment.

Thus, JavaBIP addresses the major coordination issues by providing strong modularity, which, in its turn, enables maximal reusability of components and facilitates the management of product variants. The separation of functional and coordination aspects greatly reduces the burden of system complexity. Finally, integration with the BIP framework (cf. Section 2), allows the use of existing tools [6, 7] for deadlock-detection and model checking, ensuring the correctness of generated systems.

1.1. The BIP Component Framework

JavaBIP implements the BIP coordination mechanism [4]. BIP is a framework for component-based design of correct-by-construction applications. It provides a simple, but powerful mechanism for coordination of concurrent components 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 [8]. 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 [9]. 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. Paper Contribution and Structure

In [10], we have published a preliminary, proof-of-concept implementation of our approach focusing on the adaptation of the BIP coordination primitives in the Java software-engineering context. The main difference of that implementation from the classical BIP was the addition of spontaneous transitions, which allow components to update their state based on the events occurring in their environment (cf. Section 3).

This paper presents a new implementation, which has been almost completely redesigned and extended as follows. First of all, in order to make JavaBIP applicable to practical systems, we have extended the behavior specification and the coordination mechanisms to explicitly handle data and data transfer between components. In order to give developers full control of data exposure and use within the interactions among components, we have implemented a simplified form of data transfer, which, however, has the same expressive power as in the classical BIP.

DOI: 10.1002/spe

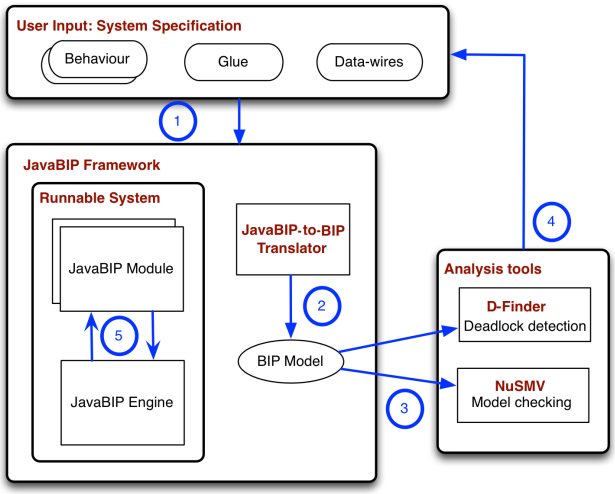


Figure 1. JavaBIP design workflow.

In order to implement the data transfer and facilitate future extensions of the JavaBIP framework, we have developed a new modular and extensible engine, consisting of a symbolic kernel and a *coordinator stack*. At each execution cycle, the kernel determines the next interaction by solving the combined system constraints. Each coordinator is responsible for 1) encoding a particular type of constraints into a form that the kernel can process and 2) interpreting the relevant part of the solution provided by the kernel to drive the component execution. We implemented two coordinators: the Glue Coordinator, implementing the BIP coordination mechanism, and the Data Coordinator, controlling data transfer among components.

We provide a formalization of the semantic model underlying the JavaBIP implementation. Although this formalization is based on the previous work on BIP [4, 8, 11], it is adapted and extended to reflect the specificities of JavaBIP. Finally, we have designed new examples that we use to illustrate the JavaBIP modelling concepts and evaluate the overhead induced by the engine.

The rest of the paper is structured as follows. Section 2 presents the JavaBIP design workflow. Section 3 provides examples of JavaBIP models, explaining how the proposed design methodology can be applied in practice. Section 4 defines the formal semantic model underlying the JavaBIP design.[†] Section 5 presents the annotations and XML constructs used to design JavaBIP specifications. Section 6 describes the implemented software architecture and presents performance results. Section 7 presents the related work. Section 8 summarizes the results and future work directions.

2. JAVABIP DESIGN WORKFLOW

Figure 1 shows the steps of the JavaBIP design flow. In the step ①, the *system specification* is designed, consisting of the following annotated Java classes and XML configuration files:

- A *behavior 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.

[†]The formal material in Section 4 can be omitted by readers interested only in understanding how to use the JavaBIP framework.

Prepared using speauth.cls
DOI: 10.1002/spe

- The *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 optional analysis loop starts in step ②, where the system specification is automatically translated into an equivalent model of the system expressed in the BIP language.[‡] This model can then be verified for deadlock freedom or other safety properties (we provide examples in Section 3.1 and Section 3.2), using DFinder [6], ESST or nuXmv [7] (step ③). Other analyses can be performed using any tool for which a model transformation from BIP is available. If the required safety properties are not satisfied by the model, the specification can be refined by the developers (step ④) and analyzed anew. Finally, when developers are satisfied with the design, the refined specification can be executed (step ⑤). Steps ②, ③, and ④ are optional and can be repeated several times.

As shown in Figure 1, the *runnable system* consists of two major parts: the *engine* and several *modules*, one for each component to be coordinated. Each module is composed of a dedicated executor, a behavior specification and the corresponding functional code of the component (cf. Figure 15). The executor has access to the behavior specification of the component and uses it to drive the module execution. The behavior specification of each component along with the glue and data-wire specifications are provided to the engine. The engine orchestrates the overall execution of the system by deciding which component transitions must be executed at each cycle. It then notifies the executors of the selected transitions and they make the corresponding calls to the functional code.

3. JAVABIP BY EXAMPLES

In this section, we provide three examples illustrating the modeling concepts of JavaBIP. The Camel routes example (Section 3.1) illustrates the basic notions, i.e. the FSMs, the three kinds of transitions and simple connectors. The Publish-Subscribe server example (Section 3.2) illustrates use of notifications to encode Actor-like asynchronous communication. Finally, Trackers and Peers (Section 3.3) is a toy example intentionally designed with a complex coordination pattern involving multiparty interaction. These examples are also used in the rest of the paper to illustrate the proposed annotation language and, in Section 6.3, to evaluate the overhead incurred by using the JavaBIP framework. In particular, the complexity of the coordination pattern of the Trackers and Peers example will allow us to subject the JavaBIP engine to a stress test under high coordination load and evaluate the practical limitations of the current implementation.

FSM transitions can be of three types: *enforceable, spontaneous* and *internal*. Enforceable transitions are controlled by the engine. At each execution cycle, executors inform the engine about enforceable transitions offered by the components in their current state. The engine decides which of these should be executed and notifies the executors of its decision. 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.

Interaction models can be represented in many equivalent ways. Among these are connectors [8] and Boolean formulas on variables representing port participation in interactions [12]. Connectors are most appropriate for graphical design and interaction representation, whereas Boolean formulas

[‡]We have developed a prototype of the JavaBIP-to-BIP transformation tool. However, its presentation is not in the scope of this paper.

DOI: 10.1002/spe

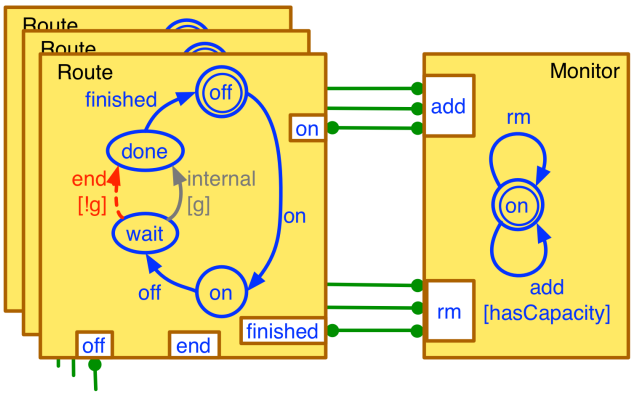


Figure 2. JavaBIP models of three routes and a monitor.

are most appropriate for manipulation and efficient encoding. In this paper, we use the graphical notation in the figures illustrating the examples. Equivalent notation based on Boolean macros (cf. Section 5) is used to define the allowed synchronizations in the glue specification.

In Figures 2, 5 and 8, the initial states of all FSMs are denoted by a double circle; Boolean guards on transitions are shown in square brackets; enforceable transitions are shown with solid blue arrows, spontaneous transitions with dashed red arrows and internal transitions with solid grey arrows.

3.1. Camel routes

A Camel route [1] 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.

Consider the Route and Monitor models, shown in Figure 2. Our goal is to limit the number of routes running simultaneously to ensure that the available memory is sufficient for the safe functioning of the system. To achieve this, we introduce an additional monitor component. The behavior specifications of the Route and Monitor component types are shown in Figures 3 and 4 respectively. The precise syntax is explained in Section 5.1.

The Route model, shown in the left-hand side of Figure 2, has four states: `off`, `on`, `wait` and `done`. Its initial state is `off`. When the route is at state `off`, it can start working by executing the `on` transition. Respectively, when the route is at state `on`, it can suspend its work by executing the `off` transition. The `on` and `off` transitions are both enforceable (Figure 3: lines 3–4) and are associated with the `resumeRoute` and `suspendRoute` methods of the Camel API (Figure 3: lines 19–27).

Following the call to `suspendRoute` associated with the transition `off`, the route moves to the state `wait`. At this point, if the route has finished processing the previous data batch, it can be suspended immediately—represented by the internal transition to the `done` state (Figure 3: lines 32–33). Otherwise, the internal transition is disabled. Instead, to move to state `done`, the route has to wait for the processing termination event, associated with the spontaneous transition `end` (Figure 3: lines 29–30). The guard `g` (Figure 3: lines 38–42) is used to check whether the route has finished processing.

The Monitor model, shown in the right-hand side of Figure 2, has only one state and two enforceable transitions: `add` (Figure 4: lines 16–20) for adding running routes and `rm` (Figure 4: lines 22–25) for removing them. The `add` transition has the guard `hasCapacity` (Figure 4:

DOI: 10.1002/spe

```
1 @Ports({
2   @Port(name = "end", type = PortType.spontaneous),
3   @Port(name = "on",  type = PortType.enforceable),
4   @Port(name = "off", type = PortType.enforceable),
5   @Port(name = "finished", type = PortType.enforceable)
6 })
7 @ComponentType(initial = "off", name = "Route")
8 public class Route implements CamelContextAware {
9
10  private CamelContext camelContext;
11  private String routeId;
12  private int deltaMemory = 100; // Dummy value, for the sake of simplicity
13
14  public Route(String routeId, CamelContext camelContext) {
15    this.routeId = routeId;
16    this.camelContext = camelContext;
17  }
18
19  @Transition(name = "on", source = "off", target = "on")
20  public void startRoute() throws Exception {
21    camelContext.resumeRoute(routeId);
22  }
23
24  @Transition(name = "off", source = "on", target = "wait")
25  public void stopRoute() throws Exception {
26    camelContext.suspendRoute(routeId);
27  }
28
29  @Transition(name = "end", source = "wait", target = "done", guard = "!g")
30  public void spontaneousEnd() {} // "!g" in the guard above means "not g"
31
32  @Transition(name = "", source = "wait", target = "done", guard = "g")
33  public void internalEnd() {}
34
35  @Transition(name = "finished", source = "done", target = "off")
36  public void finishedTransition() {}
37
38  @Guard(name = "g")
39  public boolean isFinished() {
40    return camelContext.getInflightRepository().
41      size(camelContext.getRoute(routeId).getEndpoint()) == 0;
42  }
43
44  @Data(name = "deltaMemoryOnTransition",
45    accessTypePort = AccessType.allowed, ports = { "on", "finished" })
46  public int deltaMemoryOnTransition() {
47    return deltaMemory;
48  }
49 }
```

Figure 3. Annotations for the Route component type.

lines 27–30) that checks whether the available memory limit of the system, defined through the constructor of the MemoryMonitor class (Figure 4: lines 12–14), is sufficient for adding more running routes.

The complete system consists of several routes and one monitor. The Route model is the same for all routes and the monitor is connected to all of them in the same manner. The port `on` of each route component must synchronize with the port `add` of the monitor. This means that when a route component is executing the `on` transition, the monitor component must execute the `add` transition simultaneously. Thus, if the available memory capacity is not sufficient, the `on` transition is blocked. Since the `add` port of the monitor is connected to the `on` ports of several different routes by binary connectors, it must only synchronize with one of them at a time. Similarly, transition `finished` of

DOI: 10.1002/spe

```
1 @Ports({
2  @Port(name = "add", type = PortType.enforceable),
3   @Port(name = "rm", type = PortType.enforceable)
4 })
6 @ComponentType(initial = "on", name = "MemoryMonitor")
7 public class MemoryMonitor {
8
9   final private int memoryLimit;
10  private int currentCapacity = 0;
11
12  public MemoryMonitor(int memoryLimit) {
13    this.memoryLimit = memoryLimit;
14  }
15
16  @Transition(name = "add", source = "on", target = "on",
17              guard = "hasCapacity")
18  public void addRoute(@Data("memoryUsage") Integer deltaMemory) {
19    currentCapacity += deltaMemory;
20  }
21
22  @Transition(name = "rm", source = "on", target = "on")
23  public void removeRoute(@Data(name="memoryUsage") Integer deltaMemory) {
24    currentCapacity -= deltaMemory;
25  }
26
27  @Guard(name = "hasCapacity")
28  public boolean hasCapacity(@Data("memoryUsage") Integer memoryUsage) {
29    return currentCapacity + memoryUsage < memoryLimit;
30  }
31 }
```

Figure 4. Annotations for the Monitor component type.

each route must be synchronized with the transition `rm` of the monitor. Notice that the specification of synchronizations between the `on-add` transitions and `rm-finished` transitions is not part of the behavior specification presented in Figures 2 and 4. These synchronizations are included in the glue specification presented in Figure 13.

At each execution cycle, the monitor decides whether there is sufficient amount of memory in the system to add another route. To do that, data are being exchanged between the non-running routes and the monitor. Every route has a value of how much memory it consumes if it resumes working (Figure 3: lines 44–48) and sends this to the monitor. The monitor then decides by computing the `hasCapacity` guard value and informs the JavaBIP engine of its decision. Data exchange between the routes and the monitor happens if the `on-add` synchronization can be executed, i.e., some of the routes are at state `off`. Additionally, data exchange occurs before the `finished-rm` synchronization is executed: the corresponding route tells the monitor how much memory it will release upon suspending its execution.

Notice that the access control functionality implemented in this example, can also be implemented in an actor-based framework, through a synchronization on a future. Indeed, by implementing the monitor as an actor capable of serving the request `hasCapacity`, it is sufficient to send such a request before resuming a route and store the returned Boolean yes-or-no value in a future, then immediately consulting this future as part of a branching condition. This would block the route activation until the reply from the monitor is available, effectively achieving a synchronization between the route and the monitor. The advantage of the JavaBIP approach is that—contrary to the solution using a Boolean future—it does not require the synchronization to be hardcoded in the route specification: in the ideal situation, where infinite memory is available, *exactly the same* BIP specification of the routes can be used without the monitor and the corresponding glue. Similarly, if, at a later stage in the project or for the purpose of debugging, the developer needed to introduce a logger component to keep track of route management operations, this could be achieved simply

DOI: 10.1002/spe

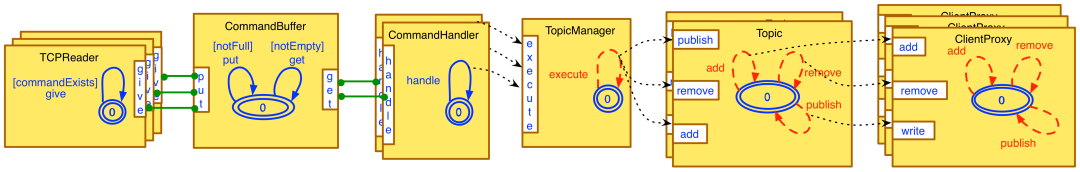


Figure 5. JavaBIP models of a publish-subscribe server.

by defining the corresponding BIP specification and replacing binary synchronizations (Figure 2) by ternary ones. As will be explained in Section 5.2, this latter operation only involves editing an XML configuration file, which can even be performed by a system administrator, without requiring an intervention by a developer.

The complete model of this example is deadlock free and satisfies the following safety property: *'A route cannot be added unless sufficient memory capacity is available'*. Notice that this property is enforced on an arbitrary number of routes by synchronizing them in the predefined manner with an additional Monitor component. The Monitor component together with the associated connectors (Figure 2) forms a pattern, called an *architecture*. In [13, 14], the authors have developed a compositional theory of architectures—illustrated by a large case study in [15]—that allows the definition and composition of such patterns to enforce arbitrary combinations of safety properties.

3.2. *Publish-Subscribe server*

In this section, we present a JavaBIP implementation of the Publish-Subscribe server. The server manages a number of different topics to which the clients can subscribe, unsubscribe and publish messages. To that end, clients send commands, which are handled by the server in a concurrent fashion.

Consider the model in Figure 5. The solid connectors between ports denote synchronizations, i.e., the execution of the corresponding transitions must be synchronized. In all synchronizations, data are exchanged between components. The dotted arrows from transitions to ports represent asynchronous communication realized by generating events corresponding to the spontaneous transitions of the receiving components.

The server consists of components of the six types shown in Figure 5. For each client, there is a dedicated TCPReader, responsible for receiving commands. Additionally, for each client there is a dedicated ClientProxy, responsible for receiving 1) acknowledgments that the client has been added or removed from a topic and 2) messages published from other clients registered in same topics. Thus each client is modeled by two dedicated components (a TCPReader and a ClientProxy), each modeling a distinct functional aspect of the client.

Upon reception by a TCPReader, each command is forwarded to the unique CommandBuffer component through the synchronization of the `give` and `put` enforceable transitions. The guard `commandExists` of the TCPReader is used to check whether it has received a new command and the guard `notFull` of the CommandBuffer is used to check whether the buffer is not full before receiving a new command (Figure 6: lines 29–30). If both guards evaluate to *true*, the command is transferred as data to the CommandBuffer (Figure 6: line 25).

The CommandBuffer is a passive component: the responsibility for retrieving commands from the CommandBuffer belongs to CommandHandlers. This happens through the synchronization of the `handle` and `get` enforceable transitions, when the `notEmpty` guard evaluates to *true* (Figure 6: lines 21–22). There can be arbitrarily many CommandHandlers that are concurrently handling commands. The CommandHandlers asynchronously forward commands to the TopicManager, by generating the event associated to the `execute` spontaneous transition of the TopicManager (Figure 7: lines 2 & 13). Notice that the `@Data` annotation is used to define input data necessary for the processing of spontaneous events (Figure 7: line 14). This mechanism allows CommandHandlers to send commands in the form of data to the TopicManager, in a manner very similar to asynchronous message passing.

DOI: 10.1002/spe

```
1 @Ports({
2   @Port(name="put", type=PortType.enforceable),
3   @Port(name="get", type=PortType.enforceable)
4 })
5
6 @ComponentType(initial="0", name="CommandBuffer")
7 public class CommandBuffer {
8   private LinkedList<Command> commandList;
9   private int bufferSize;
10
11  public CommandBuffer(int bufferSize){
12    this.bufferSize = bufferSize;
13    this.commandList = new LinkedList<Command>();
14  }
15
16  @Transition(name="get", source="0", target="0", guard="notEmpty")
17  public void get() {
18    commandList.remove();
19  }
20
21  @Guard(name="notEmpty")
22  public boolean notEmpty() { return !commandList.isEmpty(); }
23
24  @Transition(name="put", source="0", target="0", guard="notFull")
25  public void put(@Data(name="input") Command cmd) {
26    commandList.add(cmd);
27  }
28
29  @Guard(name="notFull")
30  public boolean notFull() { return commandList.size() < bufferSize; }
31
32  @Data(name="command")
33  public Command getNextCommand() { return commandList.get(0); }
34 }
```

Figure 6. Annotations for the CommandBuffer component type.

Depending on the type of the command (Figure 7: lines 16, 20, 24), the TopicManager asynchronously triggers one of the `add`, `remove` or `publish` transitions of the corresponding Topic (Figure 7: lines 18, 22, 26). The Topic executes the commands and triggers the corresponding transitions of a ClientProxy to either send an acknowledgment to the client in the case of the subscribe/unsubscribe commands or to distribute the message to all the subscribed clients of the topic in the case of a publish command. All transitions of TopicManager, Topic and ClientProxy components are spontaneous, i.e., they are executed asynchronously upon reception of the corresponding events.

An example of the above execution is the following: A TCPReader forwards the command `subscribe epfl` via data transfer through the synchronization of the `give` transition of the TCPReader with the `put` transition of the CommandBuffer. A CommandHandler receives the command from the CommandBuffer through the synchronization of the `get` and `handle` transitions. Then, the CommandHandler forwards the command to the TopicManager. This triggers the `execute` spontaneous transition during which the client reference is retrieved from the command and transferred to the `epfl` topic. This results in the asynchronous execution of the `add` transition of the `epfl` topic. During the execution of `add` the client reference, which was received though the data transfer, is stored and the name of the topic (epfl) is forwarded to the dedicated ClientProxy asynchronously via the data transfer. The ClientProxy stores the topic name and writes an acknowledgment in the socket, while executing the `add` spontaneous transition asynchronously.

DOI: 10.1002/spe

```
1 @Ports({
2  @Port(name="execute", type=PortType.spontaneous)
3 })
4
5 @ComponentType(initial="0", name="TopicManager")
6 public class TopicManager {
7   private HashMap<String, BIPActor> topics;
8
9   public TopicManager(HashMap<String, BIPActor> topics) {
10    this.topics = topics;
11  }
12
13  @Transition(name="execute", source="0", target="0")
14  public void execute(@Data(name="value") Command c) {
15    switch (c.getId()) {
16    case SUBSCRIBE:
17      Topic topic = topics.get(c.getTopic());
18      topic.add(c.getClient());      // Generate an "add" event in the topic
19      break;
20    case UNSUBSCRIBE:
21      Topic topic = topics.get(c.getTopic());
22      topic.remove(c.getClient()); // Generate a "remove" event in the topic
23      break;
24    case PUBLISH:
25      Topic topic = topics.get(c.getTopic());
26      topic.publish(c.getClient(), c.getMessage());
27      break;                       // Generate a "publish" event in the topic
28    default:
29      break;
30    }
31  }
32 }
```

Figure 7. Annotations for the TopicManager component type.

3.3. Trackers and Peers example

The following example was initially presented in [11]. Although it is inspired by a wireless audio protocol for peer-to-peer communication, it should be noted that this example does not implement any kind of message passing (see also the discussion in Remark 4.2). Here, we provide it purely as an example of BIP model that will be used in Section 6.3 to stress-test the JavaBIP engine. Additionally, this example illustrates the use of n-ary connectors (here $n = 3$) and of trigger ports.

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.

The model for two peers and one tracker is shown in Figure 8. Peers are allowed to use at most one channel at a time. Access to channels is subject to the following registration mechanism. Every peer selects the channel it wants to use and registers through the `register` transition that is synchronized with the `log` transition of the tracker. During this synchronization, components are exchanging data. In particular, the tracker sends its identity to the peer and the peer stores it. Once registered, peers can either speak to the channel or listen to other registered peers in the channel. To ensure atomicity of each communication, every tracker enforces that 1) at most one registered component is speaking and 2) all other registered components are listening.

In the connectors enforcing the above constraints, the `broadcast` port of a tracker is a *trigger* (Section 4.1.2). This allows the `broadcast` transition to happen on its own, without requiring synchronization with transitions of other components. However, `broadcast` accepts synchronization with the `speak` and `listen` transitions. The `listen` and `speak` ports are *synchrons*, i.e., their corresponding transitions require synchronization with `broadcast`.

DOI: 10.1002/spe

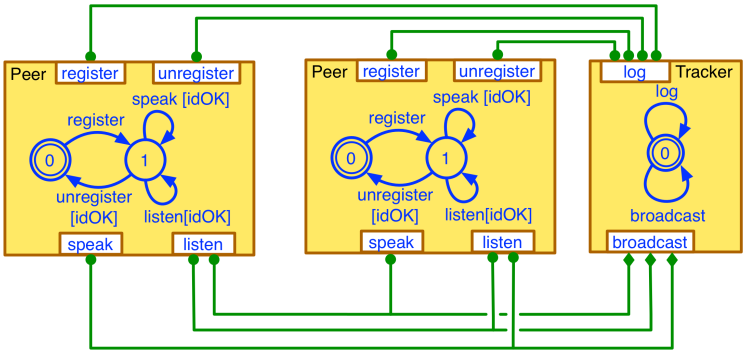


Figure 8. JavaBIP models of one tracker and two peers.

Peers can register (resp. unregister) through the `register-log` (resp. `unregister-log`) synchronization.

Notice that the interaction structure of the example is exponentially complex. In particular, the number of all possible interactions is $(2^T - 1) \cdot (2^P + P \cdot 2^{P-1} + 2 \cdot P)$, where P is the total number of peers and T is the total number of trackers.

To allow all peers to communicate with all trackers, all corresponding connectors must be present in the system. Data transfer (Section 4.2) is used to ensure that peers can interact only with the tracker they had previously been registered with: for each interaction, trackers propose their identity as data and peers use the `idOK` guard to decide with which trackers they can synchronize. Thus, all transitions of the system are enforceable and in all possible interactions (except when a tracker is broadcasting without any registered peers) data are exchanged between components.

4. THEORETICAL FOUNDATIONS

In this section, we provide the formal component and coordination model underlying the JavaBIP framework. This model extends that of BIP [4, 8] by considering three types of transitions:

- *enforceable* transitions represent the controllable behavior of the component;
- *spontaneous* transitions represent changes in the environment that affect the component behavior, but cannot be controlled;
- *internal* transitions represent computations independent of the component environment.

We consider a system of components, each represented by a Finite State Machine (FSM) extended with ports and data. An FSM is specified by its states and the guarded transitions between them. Each transition has a function and a port associated to it. In general, one port can be associated to several transitions. However, the FSM is deterministic in the following sense: there cannot be two simultaneously enabled transitions leaving the same state that are both internal or both labeled by the same port.

For the sake of clarity, we will first present the model without data, then extend it by introducing data variables and data wires used for the transfer of values among the components.

4.1. Component model without data

4.1.1. Components and glue A *component* is a Finite State Machine (FSM) given by a quadruple $C = (Q, P^e, P^s, \rightarrow)$, where:

- Q is a finite set of states,
- P^e and P^s are disjoint finite sets of, respectively, enforceable and spontaneous ports,

DOI: 10.1002/spe

- $\rightarrow \subseteq Q \times P \times Q$, with $P = P^e \cup P^s \cup \{\tau\}$ and $\tau \notin P^e \cup P^s$, is a transition relation, where the special symbol τ is used to label internal transitions.

Below, we will use the common notation, writing $q \xrightarrow{p} q'$ as a shorthand for $(q, p, q') \in \rightarrow$. We say that a port $p \in P$ is *enabled* in the state $q \in Q$ iff there exists $q' \in Q$, such that $q \xrightarrow{p} q'$.

Remark 4.1. Notice that the above definition allows non-determinism in the component behavior. For the practical implementation, we require that all components be deterministic, i.e. such that, for any $q \in Q$ and any $p \in P$, there is at most one outgoing transition from q labeled by p or, formally, $|\{q' \in Q \,|\, (q, p, q') \in \rightarrow\}| \leq 1$. While this additional assumption does not have any impact on the theoretical foundations of the JavaBIP framework, as presented in this section, it noticeably simplifies the implementation.

Let $C_i = (Q_i, P_i^e, P_i^s, \rightarrow)$, for $i \in [1, n]$,[§] be a set of components with pairwise disjoint sets $P_i = P_i^e \cup P_i^s$, i.e., $P_i \cap P_j = \emptyset$, for all $i \neq j$. We denote $P^e = \bigcup_{i=1}^n P_i^e$ and $P^s = \bigcup_{i=1}^n P_i^s$, respectively, the sets of all enforceable and spontaneous ports in the system.

Since our goal is to model interaction through synchronization among the actions performed by components, we introduce the notion of *interaction* as a non-empty set of enforceable ports $a \subseteq P^e$, such that $|a \cap P_i^e| \leq 1$, for all $i \in [1, n]$, labeling the transitions to be synchronized. Indeed, we require that any interaction contain at most one port from any given component, reflecting the fact that a component can only execute one transition at a time.

The *glue* used to compose the components is the set of interactions that are allowed in the system. Let $\gamma \subseteq 2^{P^e} \setminus \{\emptyset\}$ be such a set of interactions. The composition of components $C_1, \ldots, C_n$ with the glue γ is the component $\gamma(C_1, \ldots, C_n) = (Q, \gamma, P^s, \rightarrow)$, where $Q = \prod_{i=1}^n Q_i$ is the Cartesian product of the sets of states of the individual components and $\rightarrow$ is the set of transitions defined as follows:

- for any $i \in [1, n]$ and any spontaneous port $p \in P_i^s$,

$$(q_1, \ldots, q_i, \ldots, q_n) \xrightarrow{p} (q_1, \ldots, q_i', \ldots, q_n) \quad \Longleftrightarrow \quad q_i \xrightarrow{p} q_i' \,,$$

- similarly for internal transitions, for any $i \in [1, n]$,

$$(q_1, \ldots, q_i, \ldots, q_n) \xrightarrow{\tau} (q_1, \ldots, q_i', \ldots, q_n) \quad \Longleftrightarrow \quad q_i \xrightarrow{\tau} q_i' \,,$$

- for any interaction $a \in \gamma$,

$$(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n') \quad \Longleftrightarrow \quad \begin{cases} q_i \xrightarrow{p_i} q_i', & \text{if } a \cap P_i^e = \{p_i\} \,, \\ q_i = q_i', & \text{if } a \cap P_i^e = \emptyset \,. \end{cases}$$

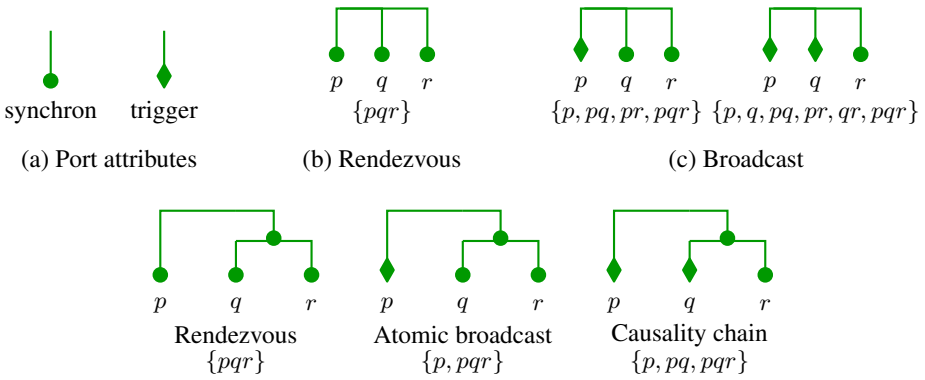
Thus, the notion of enabledness is extended from ports to interactions: an interaction $a \in \gamma$ is enabled in $\gamma(B_1, \ldots, B_n)$, only if, for each component C_i participating in a—that is, such that $a \cap P_i^e = \{p_i\}$, for some $p_i \in P_i^e$—the port p_i is enabled in C_i. Notice that the states of components that do not participate in the interaction remain unchanged. Observe also that γ defines the set of *allowed* interactions: it is possible that an interaction $a \in \gamma$ is never actually enabled in $\gamma(C_1, \ldots, C_n)$.

4.1.2. Connectors Interactions among the enforceable ports of the system components are specified by *connectors* [8]. A connector defines a set of interactions based on the synchronization attributes of the connected ports, which may be either *synchron* or *trigger* (Figure 9a):

- if all connected ports are synchrons, then synchronization is by *rendezvous*, i.e., the connector defines exactly one interaction, which comprises all its ports (Figure 9b);

[§]Here and below, we omit the index on transition relations $\rightarrow$, since it is always clear from the context.

DOI: 10.1002/spe



(a) Port attributes (b) Rendezvous (c) Broadcast

(d) Hierarchical connectors

Figure 9. BIP connectors: below each connector, we show the set of interactions it defines

- if the connector has at least one trigger, the synchronization is by *broadcast*, i.e., the connector defines the set of interactions comprised by all non-empty subsets of the connected ports containing at least one of the trigger ports (Figure 9c).

Remark 4.2. Notice that our use of the terms "broadcast" and "atomic broadcast" is very different from the meaning commonly used in distributed computing, where messages are broadcast to a number of recipients through a network. In this paper, we use the terms "rendezvous", "broadcast" and "atomic broadcast" for the sake of homogeneity with previous work on BIP (e.g. [8]), to denote different kinds of connectors as described above. The use of the term "broadcast" in [8] was inspired by previous work on Statecharts [16].

The same principle is recursively extended to hierarchical connectors, where one interaction from each subconnector is used to form an allowed interaction according to the synchron/trigger labeling of the connector nodes. For instance the causal chain connector in Figure 9d has the port p labeled as a trigger, whereas the binary broadcast subconnector q–r is labeled as a synchron. Thus the causal chain connector allows the singleton interaction p and any interaction that combines p with some interaction of the subconnector. Since the latter allows interactions q and qr, this results in three interactions allowed by the hierarchical connector: p, pq and pqr.

When, as in the examples of Section 3, there are several connectors in a system, the allowed interactions are precisely those that are defined by at least one connector.

Among the connectors in the examples of Section 3, Camel Routes and Publish Subscribe Server rely on strong binary rendezvous. The Trackers and Peers example uses weak (broadcast) synchronization defined by ternary connectors with triggers.

4.1.3. Boolean encoding of the connectors and macro notation Connectors are very convenient for graphical design and representation of interaction sets. However, their use requires the knowledge of all components (hence also all ports) in the system. In JavaBIP, we have opted for a macro notation similar to that introduced in [11]. This notation imposes synchronization constraints based on the component types, rather than on the component instances, allowing for a better separation between component and glue specification. It is based on the Boolean characterization of connectors [17], which we succinctly present below, before introducing the macro notation.

Consider a set of interactions $\gamma \subset 2^{P^e}$, where P^e is the set of all enforceable ports in the system. This set can be characterised by a Boolean predicate $\phi_\gamma \in \mathbb{B}[P^e]$ on the set of enforceable ports: an interaction $a \subseteq P^e$ belongs to γ if and only if the valuation, which assigns to each port $p \in P^e$ the value *true* if $p \in a$ and *false* otherwise, satisfies ϕ_γ. The predicate ϕ_γ is called the *characteristic predicate* of γ and is defined by putting

$$\phi_\gamma = \bigvee_{a \in \gamma} \Big(\bigwedge_{p \in a} p \wedge \bigwedge_{p \in P^e \setminus a} \overline{p} \Big) .$$

DOI: 10.1002/spe

Table I. Boolean encoding of connectors in Figure 9 (we omit the common conjunct $(a \vee b \vee c)$)

Name	Set of allowed interactions	Boolean encoding
Rendezvous	$\{pqr\}$	$(p \Rightarrow qr) \wedge (q \Rightarrow pr) \wedge (r \Rightarrow pq)$
Broadcast 1	$\{p, pq, pr, pqr\}$	$(q \Rightarrow p) \wedge (r \Rightarrow p)$
Broadcast 2	$\{p, q, pq, pr, qr, pqr\}$	$(r \Rightarrow p \vee q)$
Atomic broadcast	$\{p, pqr\}$	$(q \Rightarrow pr) \wedge (r \Rightarrow pq)$
Causality chain	$\{p, pq, pqr\}$	$(q \Rightarrow p) \wedge (r \Rightarrow q)$

Notice, however, that, for a given γ, this predicate can be equivalently written in a different form. For example, consider the left-hand side connector in Figure 9c with $P^e = \{p, q, r\}$ and $\gamma_{bdc} = \{p, pq, pr, pqr\}$. Omitting, for clarity, the conjunction operator, we then have

$$\phi_{\gamma_{bdc}} = p\,\overline{q}\,\overline{r} \;\vee\; p\,q\,\overline{r} \;\vee\; p\,\overline{q}\,r \;\vee\; p\,q\,r \equiv p\,,$$

which corresponds, indeed, to the meaning defined above: this connector allows any interaction among the ports p, q and r that contains p. Similarly, for the causality chain in Figure 9d, we have $\gamma_{cc} = \{p, pq, pqr\}$, characterised by

$$\phi_{\gamma_{cc}} = p\,\overline{q}\,\overline{r} \;\vee\; p\,q\,\overline{r} \;\vee\; p\,q\,r \equiv p \wedge (r \Rightarrow q)\,.$$

Here, the meaning of the implication $r \Rightarrow q$ is that the participation of the port r in any interaction *requires* that of the port q. Finally, observe that $\phi_{\gamma_{cc}}$ can also be equivalently rewritten as

$$\phi_{\gamma_{cc}} \equiv (p \vee q \vee r) \wedge (q \Rightarrow p) \wedge (r \Rightarrow q)\,, \tag{1}$$

which makes all such requirements explicit. Indeed the second and third conjuncts in (1) explicitate the causal dependencies visible in the connector structure, whereas the first conjunct simply states that at least one port must participate in any interaction. In [17], we have shown that the set of interactions defined by any connector can be characterised by a Boolean formula similar to (1), where the implications in each conjunct take the form

$$p \Rightarrow a_1 \vee \cdots \vee a_n\,, \tag{2}$$

with p being a port and each a_i being a conjunction of several ports. In the implication (2), we call p the *effect*, whereof $a_1, \ldots, a_n$ are the *causes*. Indeed, for p to participate in an interaction, all the ports belonging to at least one of $a_1, \ldots, a_n$ must participate. Thus, we can say that the participation of a_i, for some $i \in [1, n]$, in an interaction *is the reason why p can participate*. Table I provides the encodings of all the connectors shown in Figure 9.

Similarly to [11],[¶] we use the macro

$$p \;\; \textbf{Require} \;\; a_1; \ldots; a_n$$

to specify the constraint (2). For example the encodings of Broadcast 2 and Atomic broadcast (Table I) are, respectively,

$$r \;\; \textbf{Require} \;\; p; q \qquad \text{and} \qquad \begin{cases} q \;\; \textbf{Require} \;\; pr \\ r \;\; \textbf{Require} \;\; pq \end{cases}.$$

Notice the semicolon in the left-hand side macro.

In the commonly encountered systems, such as the examples in Section 3, most of the ports participate in very few connectors. In the constraints (1), this translates by conjuncts of the form

[¶]The notation used in [11] differs from JavaBIP in that it uses the additional **Unique** macro to specify cardinality exactly one, which in JavaBIP is expressed directly in the **Require** macro. Additionally, in [11] it is not possible to express exact cardinalities larger than one.

DOI: 10.1002/spe

$p \Rightarrow \mathit{false}$,[‖] meaning that the port p cannot participate in any interaction allowed by the connector. However, adding such conjuncts explicitly for all ports that do not participate in a connector is rather tedious. Furthermore, it is often convenient to define the constraints for a subsystem, independently of the way it will be used. In such case, some ports of the system, which will not participate in the defined interactions, are not yet known. Hence, one needs a notation to specify that only a certain set of ports can participate in an interaction that contains p, other ports being implicitly excluded. This is achieved by the macro

$$p \ \textbf{Accept} \ a\,, \qquad \text{which formally means} \qquad p \Rightarrow \bigwedge_{\substack{q \in P^e \setminus a \\ q \neq p}} \overline{q}\,.$$

The macro Accept is defined for a single port, therefore for a port r to participate in the interaction that contains p, it must be accepted by every other port participating in the interaction.

4.1.4. Macro notation based on component types In the previous subsection, we have introduced the Require-Accept macro notation for the definition of glue for a system consisting of a given set of components. Similarly to [11], JavaBIP relies on component types, rather than on component instances for the definition of glue. Although this approach is less expressive—it only allows the definition of regular glue, which imposes the same synchronization constraints on all instances of a given component type—, it provides better separation between glue and component specification. It is sufficient for practical purposes: firstly, finer specification can be obtained by considering component subtypes and, secondly, data variables and transfer (see Section 4.2) can be used by a component to remember the identities of other components it interacted with and use them to restrict subsequent interactions (e.g., continue interacting with the same component as in the previous cycle).[**]

Let $\mathcal{T}$ be a set of component types. Each *component type* represents a set of component instances with identical interfaces and behavior. For a component type $T \in \mathcal{T}$, we will write $C : T$ to denote a component C of type T; we denote $T.p$ the *port type* p, i.e., a port belonging to the interface of the type T, and $C.p$, for a component $C : T$, the *port instance* of the type $T.p$; finally, we denote $C.P^e$ the set of all enforceable ports of the component C.

We are now in position to extend the Require-Accept macro notation to component types. Let $T^1, T^2 \in \mathcal{T}$ be component types and let $C_i^1 : T^1$ and $C_j^2 : T^2$ (with $i \in [1,n]$, $j \in [1,m]$) be the corresponding component instances. We define

$$T^1.p \ \textbf{Require} \ T^2.q \quad \equiv \quad \bigwedge_{i=1}^{n} \left(C_i^1.p \ \Rightarrow \ \bigvee_{j=1}^{m} \left(C_j^2.q \wedge \bigwedge_{k \neq j} \overline{C_k^2.q} \right) \right),$$

which means that, to participate in an interaction, each of the ports $C_i^1.p$ (with $i \in [1,n]$) requires the participation of *precisely one* of the ports $C_j^2.q$ (with $j \in [1,m]$). This is in contrast with the meaning used in [11], where a separate macro is used to enforce uniqueness of the cause port. Thus, we have opted for a macro notation where the cardinality of the causes is explicit: should two instances of the same port type be required, this is specified by explicitly putting the cause port type twice:

$$T^1.p \ \textbf{Require} \ T^2.q \, T^2.q \quad \equiv \quad \bigwedge_{i=1}^{n} \left(C_i^1.p \ \Rightarrow \ \bigvee_{j=1}^{m} \bigvee_{k \neq j} \left(C_j^2.q \, C_k^2.q \wedge \bigwedge_{l \neq j,k} \overline{C_l^2.q} \right) \right)$$

and so on for higher cardinalities. This choice of notation is motivated by our observation that cardinalities higher than one are very rare in practical examples.

[‖] This is equivalent to (2) with the empty list of causes.
[**] This is similar to the use of history variables in [11].

DOI: 10.1002/spe

Similarly, we define the notation for the Accept macro for component types:

$$T^1.p \ \textbf{Accept} \ T^2.q \quad \equiv \quad \bigwedge_{i=1}^{n} \left(C_i^1.p \ \Rightarrow \bigwedge_{\substack{r \in P^e \setminus \{C_j^2.q \,|\, j \in [1,m]\} \\ r \neq C_i^1.p}} \overline{r} \right),$$

$$\text{where} \quad P^e \quad = \quad \bigcup_{T \in \mathcal{T}} \bigcup_{C:T} C.P^e \,.$$

The generalization of the above definitions to more complex macros is straightforward, but cumbersome. Therefore we omit it here.

Example 4.3. To illustrate the use of the above macro notation, let us consider the Camel routes example (Section 3.1). The glue constraints imposed by the connectors in Figure 2 are specified by the following combination of macros:

`Monitor.add`	**Require**	`Route.on`	`Monitor.add`	**Accept**	`Route.on`
`Monitor.rm`	**Require**	`Route.finished`	`Monitor.rm`	**Accept**	`Route.finished`
`Route.on`	**Require**	`Monitor.add`	`Route.on`	**Accept**	`Monitor.add`
`Route.finished`	**Require**	`Monitor.rm`	`Route.finished`	**Accept**	`Monitor.rm`
`Route.off`	**Require**	–	`Route.off`	**Accept**	–

,

where the dashes '–' in the last line indicate that the port `Route.off` neither requires, nor accepts synchronization with any other port. Recall that the port `Route.end` is spontaneous. Hence, it does not have any associated glue constraints. Finally, notice that this set of macros is independent from the number of Camel routes in the system.

Example 4.4. Now, let us consider an alteration of the previous example, where we require that the Monitor component removes two routes simultaneously at the execution of the port `rm`. Thus, in the require macro for the `rm` port type, the cardinality of the causes must be equal to two. This is specified by putting the cause port type `finished` twice as shown below:

`Monitor.add`	**Require**	`Route.on`	`Monitor.add`	**Accept**	`Route.on`
`Monitor.rm`	**Require**	`Route.finished`	`Monitor.rm`	**Accept**	`Route.finished`
		`Route.finished`			
`Route.on`	**Require**	`Monitor.add`	`Route.on`	**Accept**	`Monitor.add`
`Route.finished`	**Require**	`Monitor.rm`	`Route.finished`	**Accept**	`Monitor.rm`
					`Route.finished`
`Route.off`	**Require**	–	`Route.off`	**Accept**	–

.

Notice that the accept macro for the `finished` port type has also changed, by *accepting* not only `rm` but also `finished`.

Example 4.5. Let us now consider a more general example to illustrate the expressiveness of the JavaBIP glue. Assume that there are three component types `A, B, C` with port types `a, b, c`, respectively. Through the require macros, we enforce the following three constraints: 1) `A.a` requires synchronization with two instances of `B.b`; 2) `B.b` requires synchronization either with a) a single instance of `A.a` and a single instance of `C.c` or b) just two instances of `C.c`; 3) `C.c` does not require synchronizations with other ports, however it accepts synchronizations with any possible combination of ports `A.a, B.b, C.c`:

DOI: 10.1002/spe

A.a	**Require**	B.b B.b	**Accept**	A.a B.b C.c
B.b	**Require**	A.a C.c ; C.c C.c	**Accept**	A.a B.b C.c
C.c	**Require**	–	**Accept**	A.a B.b C.c .

Notice that by the combination of the first two require macros, a synchronization involving exactly an instance of `A.a` and two instances of `B.b` is not allowed, since `B.b` requires at least one instance of `C.c` to also participate in the synchronization.

Example 4.6. The set of interactions allowed by the connectors in the Trackers and Peers example (Section 3.3) can be specified as follows (notice the semicolon in the require constraint for `Tracker.log`):

Peer.speak	**Require**	Tracker.broadcast		
Peer.speak	**Accept**	Tracker.broadcast	Peer.listen	
Peer.listen	**Require**	Tracker.broadcast		
Peer.listen	**Accept**	Tracker.broadcast	Peer.speak	Peer.listen
Peer.register	**Require**	Tracker.log		
Peer.register	**Accept**	Tracker.log		
Peer.unregister	**Require**	Tracker.log		
Peer.unregister	**Accept**	Tracker.log		
Tracker.broadcast	**Require**	–		
Tracker.broadcast	**Accept**	Peer.speak	Peer.listen	
Tracker.log	**Require**	Peer.register ;	Peer.unregister	
Tracker.log	**Accept**	Peer.register	Peer.unregister	.

4.2. Extension of the model with data

4.2.1. Components with data Data transfer is essential to allow information flow between components. For each component we consider two types of data:

- *input data* that the component receives from its environment (including other components);
- *output data* that the component provides to other components.

To each type of data we associate the corresponding set of variables. For simplicity, we assume in this section that all variables have the same domain $\mathcal{D}$. In our implementation, variables can have any type that can be defined in a Java program.

In the presence of data, the state of a component is defined by the combination of the control location and the valuation of its output variables. Since the input data are provided by the environment, they do not contribute to the state of the component. This is analogous to the combination of the program counter and the valuation of variables in the semantics of programming languages. In the absence of data (Section 4.1), the state and the control location coincide. Therefore, for the sake of presentation uniformity, we will continue referring as "state" to the control location of the FSM underlying the component; we will refer as "complete state" to the combination of the control location and the valuation of output variables.

Let us first introduce some additional notation. Assume that X^{in} and X^{out} are two given sets of, respectively, input and output variables of a component. We denote

$\mathbb{B}[X^{in}, X^{out}]$,	the set of Boolean predicates on input and output variables,
$\mathbb{E}[X^{in}, X^{out}]$,	the set of assignment expressions of the form $Y := e(X)$, with $X \subseteq X^{in} \cup X^{out}$ and $Y \subseteq X^{out}$.

For a guard $g \in \mathbb{B}[X^{in}, X^{out}]$ or an expression $f \in \mathbb{E}[X^{in}, X^{out}]$, we denote $in(g), in(f) \subseteq X^{in}$ and $out(g), out(f) \subseteq X^{out}$ the corresponding sets of input and output variables used in g and f.

A *component with data* is a tuple $C = (Q, P^e, P^s, X^{in}, X^{out}, d, \rightarrow)$, where:

DOI: 10.1002/spe

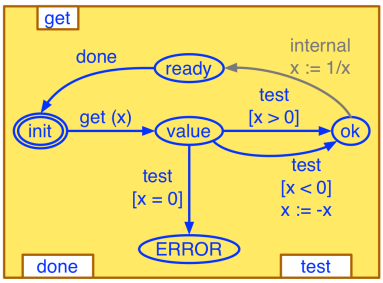


Figure 10. Example of a non-deterministic component with data

- Q is a finite set of states,
- P^e and P^s are disjoint finite sets of, respectively, enforceable and spontaneous ports,
- X^{in} and X^{out} are disjoint finite sets of, respectively, input and output variables,
- $d : P^e \to 2^{X^{out}}$ is a *data access mapping*, associating to each enforceable port $p \in P^e$ the set of output variables that are accessible through p;
- $\to \subseteq Q \times P \times \mathcal{G} \times \mathcal{F} \times Q$ is a transition relation (as above, we write $q \xrightarrow{p,g,f} q'$ as a shorthand for $(q, p, g, f, q') \in \to$), where
  - $\mathcal{G} = \mathbb{B}[X^{in}, X^{out}]$ is a set of *guards*,
  - $\mathcal{F} = \mathbb{E}[X^{in}, X^{out}]$ is a set of *update expressions*,
  - $P = P^e \cup P^s \cup \{\tau\}$ with $\tau \notin P^e \cup P^s$,

  such that $in(f) = in(g) = \emptyset$, for any internal transition $q \xrightarrow{\tau,g,f} q'$.

Example 4.7. This definition of components with data can be illustrated by the listings in Figure 3 and 4. In Figure 3, there is only one `@Data` annotation (lines 44–45) associated to a method. It defines the output variable `deltaMemoryOnTransition` (see Section 5.1 for the detailed presentation of the `@Data` annotation). Thus, we have $X^{in} = \emptyset$ and $X^{out} = \{\texttt{deltaMemoryOnTransition}\}$. Furthermore, the fields `accessTypePort` and `ports` indicate that this output variable can be accessed through ports `on` and `finished`. Thus the data access mapping is given by:

$$d = \left\{ \begin{array}{r} \texttt{on} \mapsto \{\texttt{deltaMemoryOnTransition}\}, \\ \texttt{off} \mapsto \emptyset, \\ \texttt{finished} \mapsto \{\texttt{deltaMemoryOnTransition}\} \end{array} \right\}.$$

The `@Data` annotations in the listing in Figure 4 (lines 18, 23 and 28) are associated to method parameters. They define three utilisations of the same input variable `memoryUsage`. Thus, we have $X^{in} = \{\texttt{memoryUsage}\}$, $X^{out} = \emptyset$ and, since there are no output variables, the data access mapping is constant: $d(p) = \emptyset$, for all ports $p \in P^e$.

Remark 4.8. Similarly to Remark 4.1, the above definition of components with data allows non-deterministic behavior. In the practical implementation, we require that all components be deterministic, i.e. we require that for any port $p \in P$ and any *complete* state (q, v), with state $q \in Q$ and valuation of variables $v : X^{in} \cup X^{out} \to \mathcal{D}$, we have $\left|\{(g, f, q') \in \mathcal{G} \times \mathcal{F} \times Q \,|\, q \xrightarrow{p,g,f} q',\ g(v) = true\}\right| \leq 1$.

Notice that, as opposed to Section 4.1.1, in a component with data, there can be several outgoing transitions in the same state, labeled with the same port. However, we require that at most one of

DOI: 10.1002/spe

them be enabled—meaning that its guard evaluates to $true$—with any valuation of the component variables. For example, consider the component shown in Figure 10. Although there are three outgoing transitions in state `value` labelled with the same port `test`, this component is, indeed, deterministic, since only one of these transitions can be enabled, for any value of x.

The input variables do not have persistent values—they represent the arguments of transition guards and update expressions, and have to be assigned a value provided by the environment every time, when the corresponding guard or update expression must be evaluated. Hence, indeed, the sets X^{in} and X^{out} are disjoint.

As mentioned above, at any given execution cycle, the complete state of the component is formed by the combination of its current control location and the valuation of its output variables. The component can change its complete state by executing a transition only if its environment provides all the necessary inputs—i.e., the input data required by the guard and the update expression of the transition—and the guard is satisfied. Upon executing the transition the values of the output variables are modified using the update expression. Notice that the values of the output variables that a component provides are those before the execution of the transition. Furthermore, these need not be the same as the output variables updated by the transition. In other words, for a transition $q \xrightarrow{p,g,f} q'$, with $p \in P^e$, there is no *a priori* relation between $d(p)$ and $out(f)$.

We denote $\widetilde{C} = (Q, P^e, P^s, \rightarrow_{da})$, with $q \xrightarrow{p}_{da} q'$ iff $q \xrightarrow{p,g,f} q'$, for some $g \in \mathcal{G}$ and $f \in \mathcal{F}$, the component obtained by abstracting the data from the component with data $C = (Q, P^e, P^s, X^{in}, X^{out}, d, \rightarrow)$.

4.2.2. Composition Let $C_i = (Q_i, P_i^e, P_i^s, X_i^{in}, x_i^{out}, d_i \rightarrow)$, for $i \in [1, n]$, be a set of components with data with pairwise disjoint sets of ports $P_i = P_i^e \cup P_i^s$ and variables $X_i = X_i^{in} \cup X_i^{out}$, i.e., $P_i \cap P_j = X_i \cap X_j = \emptyset$, for all $i \neq j$. We denote $P^e = \bigcup_{i=1}^n P_i^e$ and $P^s = \bigcup_{i=1}^n P_i^s$, respectively, the sets of all enforceable and spontaneous ports in the system; $X^{in} = \bigcup_{i=1}^n X_i^{in}$ and $X^{out} = \bigcup_{i=1}^n X_i^{out}$, respectively, the sets of all input and output variables.

The glue defining the composition of components with data comprises, in addition to a set of interactions $\gamma \subseteq 2^{P^e}$, the *data wire* relation $\delta \subseteq X^{in} \times X^{out}$, associating input and output variables in the system.

We say that the system is *closed* if, for each interaction allowed by γ, all input variables have corresponding output variables. Denoting, for an interaction $a = \{p_i \,|\, i \in I\} \in \gamma$, $in(a) = \bigcup_{i \in I} \big(in(g_i) \cup in(f_i)\big)$ and $out(a) = \bigcup_{i \in I} d(p_i)$, this can be formally defined as follows: for any set of component transitions $\{q_i \xrightarrow{p_i, g_i, f_i} q_i' \,|\, i \in I\}$, holds the property

$$\forall x \in in(a), \big(\exists x' \in out(a) : (x, x') \in \delta\big).$$

A given interaction is possible in the composed system $\delta\gamma(C_1, \ldots, C_n)$ if and only if the same interaction would be possible in the data-abstract system $\gamma(\widetilde{C}_1, \ldots, \widetilde{C}_n)$ and, for each of the involved components, all the necessary inputs—i.e., the input data required by the guard and the update expression of the corresponding transition—are available and the guard is satisfied.

Formally, the composition is defined as $\delta\gamma(C_1, \ldots, C_n) = (Q, \gamma, P^s, X^{in}, X^{out}, d \rightarrow)$, where $Q = \prod_{i=1}^n Q_i$ is the Cartesian product of the sets of states of the individual components, the data access mapping is defined, for any interaction $a = \{p_i \,|\, i \in I\}$, by letting

$$d(a) = \bigcup_{i \in I} d_i(p_i),$$

and $\rightarrow$ is the set of transitions defined as follows:

- for any $i \in [1, n]$ and any spontaneous port $p \in P_i^s$,

$$(q_1, \ldots, q_i, \ldots, q_n) \xrightarrow{p,g,f} (q_1, \ldots, q_i', \ldots, q_n) \quad \Longleftrightarrow \quad q_i \xrightarrow{p,g,f} q_i',$$

DOI: 10.1002/spe

- similarly for internal transitions, for any $i \in [1, n]$,

$$(q_1, \ldots, q_i, \ldots, q_n) \xrightarrow{\tau, g, f} (q_1, \ldots, q_i', \ldots, q_n) \iff q_i \xrightarrow{\tau, g, f} q_i',$$

- for any interaction $a = \{p_i \mid i \in I\} \in \gamma$ and any mapping $\delta' : in(a) \to out(a)$, such that $\big(x, \delta'(x)\big) \in \delta$, for any $x \in in(a)$,

$$(q_1, \ldots, q_n) \xrightarrow{a, g, f} (q_1', \ldots, q_n') \iff \begin{cases} q_i \xrightarrow{p_i, g_i, f_i} q_i', & \text{if } i \in I, \\ q_i = q_i', & \text{if } i \notin I, \\ g = \bigwedge_{i \in I} \widetilde{g}_i, \\ f = \big(\widetilde{f}_i\big)_{i \in I}, \end{cases}$$

  where $\widetilde{g}_i$ and $\widetilde{f}_i$ denote the expressions obtained, respectively, from g_i and f_i, by substituting each input variable x by the corresponding output variable $\delta'(x)$; and $\big(\widetilde{f}_i\big)_{i \in I}$ denotes the combined execution of the update expressions $\widetilde{f}_i$.

Notice that, in the definition of f, all the assignments are executed in parallel using the previous values provided as inputs to the component update expressions. Hence, the order of updates is irrelevant.

The presentation in this section has provided the theoretical foundations for the implementation of the JavaBIP Executor class and Engine, on one hand, and the component and glue specifications, on the other hand. Indeed, the Executor and the Engine (Section 6) implement, respectively, the semantics of components (Section 4.2.1) and glue (Section 4.2.2). In Section 5, we present the various syntactic constructs used to define JavaBIP systems. Java annotations used for the definition of BIP Specifications (Section 5.1) are based on the component model defined in Sections 4.1.1 and 4.2.1. The XML formats used for the definition of glue constraints (Section 5.2) and data wires (Section 5.3) directly mirror the corresponding formalizations in Sections 4.1.4 and 4.2.2. Short of a formal proof, this close link between the formal model and the implementation provides a high degree of confidence in the correctness of the framework. Furthermore, it enables the JavaBIP-to-BIP transformation, for the connection with the various analysis tools available in the BIP tool-set as discussed in Section 2.

5. SYSTEM SPECIFICATION

The following subsections present the constructs used to provide behavior, glue and data-wire specifications (Figure 1 step ①). We refer to the examples of Section 3 to illustrate these constructs.

5.1. Behavior specification

Developers must specify component behavior through FSMs extended with ports and data. An FSM has states and guarded transitions between them. Each transition has a method and a port associated with it. Although, in general, one port can be associated with several transitions, such transitions must have different origin state. In other words, FSMs are deterministic: there cannot be two transitions leaving the same state and labeled by the same port.

The Behavior specification can be provided via annotations associated with class, method and parameter declarations. To write a Behavior specification, developers must use the following annotations:

- `@ComponentType`: Annotates a Java class. Declares a component type by specifying its name and the initial state of the underlying FSM (Figure 7: line 5).
- `@Port`: Annotates a Java class. Declares a port by specifying its name and type—"spontaneous" or "enforceable" (Figure 7: line 2).

DOI: 10.1002/spe

- `@Ports`: Annotates a Java class. Groups all `@Port` annotations associated to a given component type (Figure 7: line 1).
- `@Guard`: Annotates a method returning a Boolean value. Declares that the method can be used as part of a transition guard, by specifying the guard name (Figure 6: line 21).
- `@Transition`: Annotates a method returning void. Declares an FSM transition, by specifying the name of the corresponding port, the source and the target states, and the guard, which is a Boolean expression on the guard names declared with the `@Guard` annotation (Figure 6: line 16). Guard expressions can be defined using parenthesis and three logical operators: negation '!', conjunction '&' and disjunction '|'.

  The type of the transition is defined by that of the port it is labeled with (Figure 6: lines 2–3 and Figure 7: line 2). Internal transitions are specified by leaving the transition name empty (Figure 3: line 32).
- `@Data`: Annotates a non-void method or a method parameter. Defines the data required (input) or provided (output) by the component:
  - *input datum*, when associated with a parameter of a guard or transition method (Figure 7: line 14);
  - *output datum*, when associated with a method returning a value (Figure 6: line 32).

  `@Data` annotations always have the field `name`. Data names are used to establish connections (called *data wires*—see Section 5.3) between inputs and outputs provided by the application components. When the `@Data` annotation is used to define an output datum, it has two additional fields: `accessTypePort` (`allowed`, `disallowed` or `any`) and `ports`. The latter is a list of ports of the component in question, which is used to specify how this datum can be accessed, based on the value of the `accessTypePort` field:
  - `allowed` means that the datum can be accessed only through the ports in `ports`;
  - `disallowed` means that the datum can be accessed only through the ports *not* in `ports`;
  - `any` means that the datum can be accessed at any execution point—the list of ports must be empty.

Notice that the output values are provided by the `@Data`-annotated methods. The methods associated to transitions do not return any value—they must be declared as `public void`.

The usage of data as parameters in guards allows components to disable interactions based on the data values proposed by other components. For example, in Figure 4, transition `add` (lines 16–20) depends on the guard `hasCapacity`, which requires the datum `memoryUsage` (lines 27–30). This datum is received from another component potentially participating in an interaction. If the proposed datum does not satisfy the guard, the interaction among these particular components is disabled.

5.2. Glue specification

To define the interaction model, a developer must specify the interaction constraints of the system; i.e., which ports of different components must synchronize. Interaction constraints need to be specified once for each *component type* of the system. For instance, in the Publish-Subscribe example of Figure 5, many instances of readers may exist in the system, however, the interaction model of the TCPReader component type needs to be specified only once.

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.

DOI: 10.1002/spe

```
 1 <require>
 2   <effect id="give" specType="TCPReader"/>
 3     <causes>
 4       <port id="put" specType="CommandBuffer"/>
 5   </causes>
 6 </require>
 7 <accept>
 8   <effect id="give" specType="TCPReader"/>
 9    <causes>
10       <port id="put" specType="CommandBuffer"/>
11    </causes>
12 </accept>
```

Figure 11. Glue specification for the Publish-Subscribe example (cf. Section 3.2).

```
 1 <require>
 2   <effect id="broadcast" specType="Tracker"/>
 3    <causes>
 4    </causes>
 5 </require>
 6 <accept>
 7   <effect id="broadcast" specType="Tracker"/>
 8    <causes>
 9       <port id="speak" specType="Peer"/>
10       <port id="listen" specType="Peer"/>
11    </causes>
12 </accept>
13 <require>
14   <effect id="log" specType="Tracker"/>
15    <causes>
16       <port id="register" specType="Peer"/>
17    </causes>
18    <causes>
19       <port id="unregister" specType="Peer"/>
20    </causes>
21 </require>
22 <accept>
23   <effect id="log" specType="Tracker"/>
24    <causes>
25       <port id="register" specType="Peer"/>
26       <port id="unregister" specType="Peer"/>
27    </causes>
28 </accept>
```

Figure 12. Glue specification for the Trackers-and-Peers example (cf. Section 3.3).

- *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 (cf. Figure 11). 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.

For example, the constraint in Figure 11, lines 1–6 forces the port `give` of *any* component of type `TCPReader` to synchronize with the port `put` of *some* component of type `CommandBuffer`. The constraint in Figure 11, lines 7–12 specifies that no other ports are allowed to participate in the same interaction. These two constraints essentially define binary connectors between ports `give` of each TCPReader and the port `put` of the CommandBuffer (cf. Figure 5).

DOI: 10.1002/spe

```
1  <require>
2    <effect id="on" specType="Route"/>
3    <causes>
4       <port id="add" specType="Monitor"/>
5    </causes>
6  </require>
7  <accept>
8    <effect id="on" specType="Route"/>
9    <causes>
10     <port id="add" specType="Monitor"/>
11   </causes>
12 </accept>
13 <require>
14   <effect id="finished" specType="Route"/>
15   <causes>
16       <port id="rm" specType="Monitor"/>
17   </causes>
18   </require>
19 <accept>
20   <effect id="finished" specType="Route"/>
21   <causes>
22       <port id="rm" specType="Monitor"/>
23   </causes>
24 </accept>
25 <require>
26    <effect id="add" specType="Monitor"/>
27    <causes>
28       <port id="on" specType="Route"/>
29    </causes>
30 </require>
31 <accept>
32   <effect id="add" specType="Monitor"/>
33   <causes>
34       <port id="on" specType="Route"/>
35   </causes>
36 </accept>
37 <require>
38    <effect id="rm" specType="Monitor"/>
39    <causes>
40       <port id="finished" specType="Route"/>
41    </causes>
42 </require>
43 <accept>
44   <effect id="rm" specType="Monitor"/>
45    <causes>
46       <port id="finished" specType="Route"/>
47    </causes>
48 </accept>
49 <require>
50    <effect id="off" specType="Route"/>
51    <causes>
52    </causes>
53 </require>
54 <accept>
55    <effect id="off" specType="Route"/>
56    <causes>
57    </causes>
58 </accept>
```

Figure 13. Glue specification for the Camel Routes example (Section 3.1).

The glue specification for the `broadcast` and `log` ports of the Tracker component type of Section 3.3 is shown in Figure 12. The causal constraint (Figure 12: lines 1–5) means

DOI: 10.1002/spe

```
1  <wire>
2    <from specType="TCPReader" id="readerInput"/>
3    <to specType="CommandBuffer" id="input"/>
4  </wire>
```

Figure 14. Data-wire specification for the Publish-Subscribe example (cf. Section 3.2).

that the `broadcast` ports of trackers do not require any synchronization with ports of other components. However, they accept synchronization with the `speak` and `listen` ports of the peers (Figure 12: lines 6–12). Notice that the require constraint for the port `log` has two `causes` sections, corresponding to the two independently sufficient causes (`Peer.register` and `Peer.unregister`) as discussed in Sections 4.1.3 and 3.3. In Figure 13, the complete glue specification of the Camel Routes example (Section 3.1) in XML is presented. The glue specification of the Camel Routes example with the require and accept macros was previously explained in Example 4.3. The empty `causes` sections of the constraints in lines 49–58 indicate the port `Route.off` neither requires, nor accepts synchronization with any other port.

5.3. *Data-wire 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.

The data-wire specifications must be provided in an XML file, as shown in Figure 14. In the Publish-Subscribe example of Section 3.2, the buffer component collects data from the TCPReaders that correspond to the commands sent from the clients. The data wire of Figure 14 connects the input datum of the CommandBuffer (`input`) with the output datum of the TCPReader (`readerInput`). The data transfer is finalized only if the associated transitions that require data can be executed, i.e., the corresponding guards evaluate to $true$. For instance, a TCPReader can send data only when the corresponding client sends a command and the buffer is not full, i.e., the `give-add` interaction can be executed.

6. IMPLEMENTATION

The software architecture of the *JavaBIP runnable system* is shown in Figure 15. It 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 between the engine and the modules is illustrated in Figure 15 with arrows. Information is either exchanged only once at initialization of the engine (illustrated in Figure 15 with solid arrows) or at each execution cycle (illustrated in Figure 15 with dashed arrows). Each module sends specification of its behavior. The glue and data-wire specifications are provided as XML files directly to the JavaBIP engine.

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 diverse specifications 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 15 with dashed arrows labeled `execute`. If the kernel cannot find a solution because the combined constraints are contradictory, a deadlock occurs.

DOI: 10.1002/spe

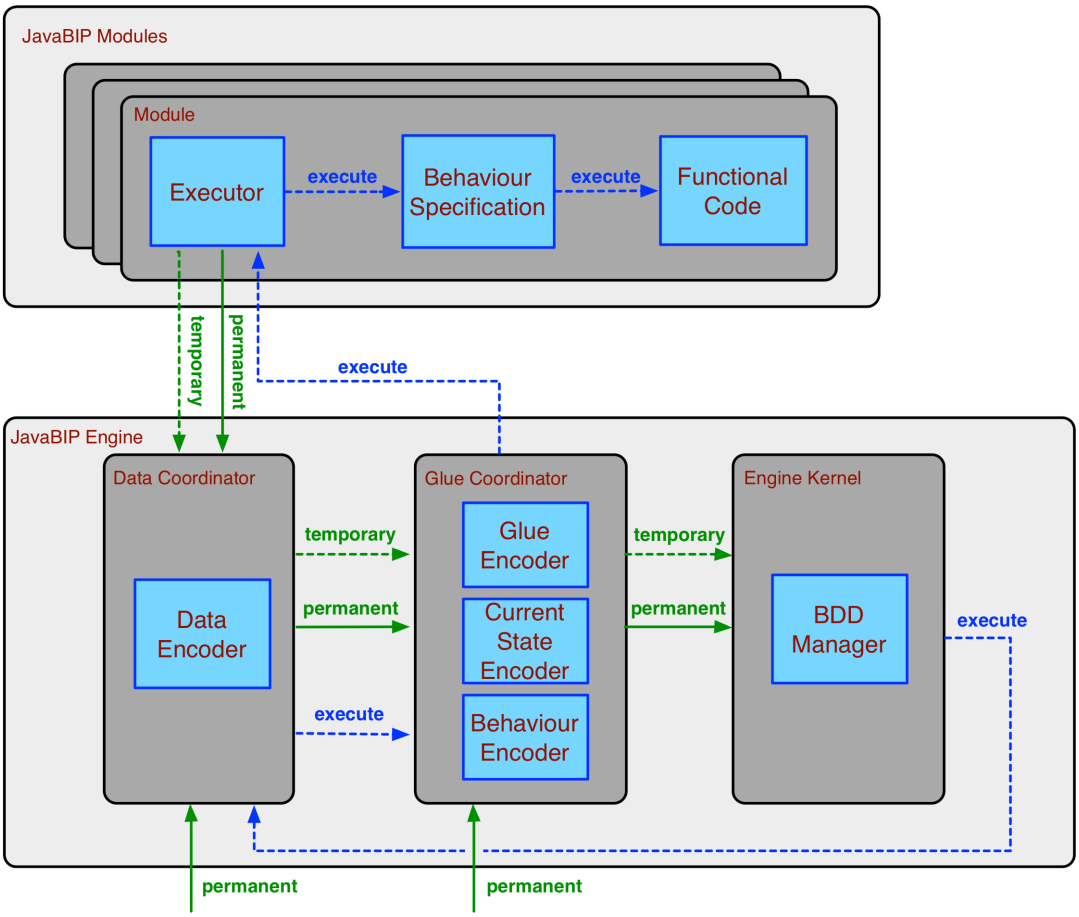


Figure 15. JavaBIP software architecture.

6.1. *JavaBIP module*

A module comprises the functional code and the behavior specification of the corresponding component (cf. Section 5.1), as well as a dedicated executor. The behavior specification contains the 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.

We have developped a generic Executor class that is instantiated in conjunction with any BIP specification, i.e., a Java class with BIP annotations, and, using Java Reflection API, drives the execution of the corresponding component. The Executor is also responsible for communicating with the JavaBIP engine to enforce the BIP protocol.

The JavaBIP engine creates a new executor instance upon registration of each module. Thus, executors are almost entirely transparent for developers. Indeed, the Executor class implements several interfaces, among which the only one that is visible to developers is the interface that provides only one method for sending spontaneous events to the executor. 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 following protocol to pick one transition to fire:

1. *Internal*: At most one internal transition can be enabled at a time (cf. Section 4). The executor fires it right away.

DOI: 10.1002/spe

2. *Spontaneous*: If no internal transitions are enabled, but the executor has received a notification corresponding to an enabled spontaneous transition, one such transition is fired. Event notifications are stored in a queue waiting to be processed. If more than one spontaneous transition can be fired, the executor picks the first from the queue.

3. *Enforceable*: If there are no internal or spontaneous transitions enabled, an enforceable one can be fired. The executor sends a list of disabled enforceable transitions to the BIP Engine and waits for a response, indicating the port to be executed. Notifications for spontaneous events that arrive at this point are not processed, but only stored in the queue. Upon receiving the response from the Engine, the Executor performs the corresponding transition.

Notice that, if no transition of any type is enabled, the component skips the cycle and waits for a notification to arrive or for a guard on one of its transitions to become $true$.

If a transition requires data for the execution, it is provided by the engine along with the port. When a transition is executed, the function associated to the transition is called using reflection and the current state of the FSM is updated. To use the transition synchronization mechanism, developers must ensure that the component states remain stable during one cycle of the engine protocol presented in Section 1.1, this means that *a component must be able to perform any transition it has announced as possible to the JavaBIP Engine*. Therefore, the component must be designed in such a way that no spontaneous transition will invalidate the ability to perform the announced enforceable transition. This is the rationale why we postpone treatment of spontaneous notifications between announcement to the engine of enabled enforceable transitions and the execution of the enforceable transition chosen by the engine.

As mentioned in Remarks 4.1 and 4.8 (Section 4), we require that FSM specifying component behavior be deterministic. The transition enabledness may depend on data evaluation, therefore the validity of this condition is, in general, undecidable and cannot be checked statically at design time. Developers are responsible for enforcing component determinism. However, the BIP Executor logs an exception when an invalid component model is detected at runtime.

6.2. JavaBIP engine

The engine comprises the kernel and a set of coordinators. The kernel combines and solves the various constraints of the system. The coordinators receive different types of information from the modules and encode the received information as Boolean constraints using dedicated encoders. These Boolean constraints are then passed to the engine kernel. We have developed two coordinators that produce different types of constraints: the Glue coordinator and the Data coordinator.

As shown in Figure 15, 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.

6.2.1. Engine kernel The kernel combines and solves the various constraints of the system. Its implementation is based on Binary Decision Diagrams (BDDs)[††] [18], which are efficient data structures to store and manipulate Boolean formulas. The kernel applies the three-step protocol presented in Section 1.1 in a cyclic manner. 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 encode information about the Behavior, Glue and data wires of the components. In Figure 15, permanent constraints are shown with arrows labeled permanent.

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

DOI: 10.1002/spe

- *Temporary constraints* that are received at each execution cycle. They encode information about the enabled transitions of the components. In Figure 15, temporary constraints are shown with arrows labeled temporary.

6.2.2. Glue coordinator The Glue coordinator manages the information about the behavior, glue and current state of the components. It encompasses three dedicated encoders (cf. Figure 15): the Behavior encoder, the Glue encoder and the 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 encoding the current states of components are temporary, hence recomputed at each execution cycle.

Each component is registered with the Glue coordinator by providing its behavior specification. Then, the Glue coordinator forwards to the Behavior encoder the lists of enforceable ports and states of each registered component. For each enforceable port, a Boolean port variable is created by the BDD manager. Similarly, for each state, a Boolean state variable is created. The behavior constraints are built, using the port and state variables.

The Behavior encoder computes the behavior BDD based on the following constraints:

1. Each component can be in one state at a time.

2. A set of enforceable ports is associated to each state. These ports can be enabled when the component is in their associated state.

3. A component may skip the cycle (execute none of the transitions and remain at the same state).

In order to avoid overloading the formulas and since glue constraints only involve enforceable ports, we omit below the superscript e used to denote enforceable ports, e.g. we write P instead of P^e. Let Q_i and P_i be the set of states and ports of a component C_i. Let $P_i^q = \{p \in P_i \mid \exists q' : q \xrightarrow{p} q'\}$. We compute the behavior BDD for component C_i, by letting

$$BDD_i^B = \bigvee_{q \in Q_i} \left(q \wedge \bigwedge_{q' \in Q_i \setminus \{q\}} \overline{q'} \wedge \bigvee_{p \in P_i^q} \left(p \wedge \bigwedge_{p \notin P_i^q} \overline{p} \right) \right) \vee \bigwedge_{p \in P_i} \overline{p}\,.$$

The system behavior BDD is computed as the conjunction of all the component behavior BDDs:

$$BDD^B = \bigwedge_{i=1}^{n} BDD_i^B.$$

The Glue encoder receives from the Glue coordinator the glue specification. The glue BDD is computed by interpreting the Require and Accept constraints as shown in Section 4.1.4, using the same Boolean port variables that have been previously created and used by the Behavior encoder. Both glue and behavior BDDs are computed once during the initialization of the system.

The Current State encoder is notified each time the Glue Coordinator is informed of a component state and its disabled ports. Ports can be disabled due to evaluation of guards. Let q be the current state and P_i^{dis} be the set of disabled ports of a component C_i. Then, the current state BDD for component C_i is computed as follows:

$$BDD_i^Q = q \wedge \bigwedge_{q' \neq q,\, q' \in Q_i} \overline{q'} \wedge \bigwedge_{p \in P_i^{dis}} \overline{p}\,.$$

The current state BDD of each component is then transferred to the engine kernel, where the conjunction of all current state BDDs is computed at each execution cycle and is further conjuncted with the Behavior and Glue BDDs.

DOI: 10.1002/spe

6.2.3. Data coordinator The Data coordinator is used on top of the Glue coordinator. Using the Data encoder, it encodes as permanent constraints the information about data wires, which connect input and output data provided by the components. In particular, at initialization phase, the Data coordinator receives the data-wire specification of the system (cf. Section 5.3). The Data coordinator queries the registered components to determine the ports that require data (input data) and the ports that provide data (output data). Then, the Data coordinator passes to the Data encoder the data wires and the pairs of ports that require and provide data. For each pair of ports, the Data encoder creates a Boolean data variable. For each data variable d and its corresponding pair of port variables p and r, the Data encoder creates the constraint

$$d \Rightarrow (p \wedge r)\,,$$

meaning that data can only be transferred along the data wire going through ports p and q if these two ports participate in the interaction. Additionally, for each port p, which has associated data variables, the Data encoder creates the following constraint:

$$p \Rightarrow \bigvee_{d \in D_p} d,$$

where D_p is the set of data variables associated to port p, meaning that These two constraints form the permanent constraint of the Data encoder, which is only computed once at initialization time. The permanent constraint is the following:

$$BDD^{D}_{perm} = \bigwedge_{d \in D} \left(\overline{d} \vee (p_d \wedge r_d)\right) \wedge \bigwedge_{p \in P,\, \text{s.t. } D_p \neq \emptyset} \left(\overline{p} \vee \bigvee_{d \in D_p} d\right),$$

where p_d and r_d are the ports connected by the data wire corresponding to the data variable d and D_p is as above.

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.

6.3. Experimental evaluation

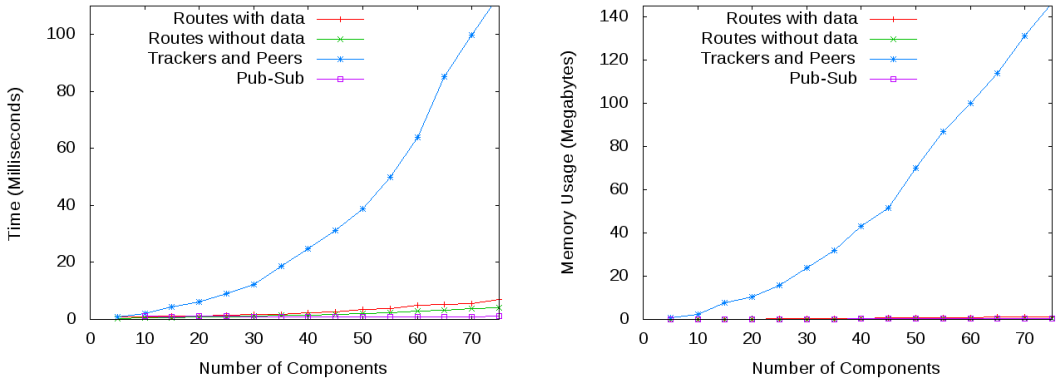
We show experimental results for four case-studies: 1) two implementations of the Camel routes example (Section 3.1), i.e., one with and one without data transfer; 2) the Trackers-and-Peers example of Section 3.3 and 3) the Publish-Subscribe example of Section 3.2. The experiments were run on an Intel Core i7-2640M CPU at 2.80GHz x 4 with 8GB RAM. The JavaBIP models of these systems are available at the JavaBIP website[‡‡].

In the Camel routes examples (with and without data), we consider $C - 1$ routes and 1 monitor, where C is the total number of components. In the Trackers-and-Peers example, there are always four times more peers than trackers. In the Publish-Subscribe example there is always one buffer, one topic-manager, and varying numbers of TCPReaders, handlers and topics. The number of client-proxies is the same as the number of TCPReaders.

Figure 16a shows the average execution time of the first 1000 engine cycles for all four examples, with the number of components ranging from 5 to 75. Figure 16b shows the peak memory usage of the BDD Manager for each of the three examples. Table II summarizes all results shown in Figures 16a and 16b. The Camel routes implementations illustrate the impact of data transfer on the performance of the engine. The behavior and interaction models of the two Camel route implementations are equivalent; in the latter, components also exchange data. Although data transfer

[‡‡] http://risd.epfl.ch/javabip

DOI: 10.1002/spe



(a) Average engine execution time.

(b) BDD Manager peak memory usage.

Figure 16. Performance diagrams

causes an increase in the execution time and memory usage of JavaBIP, the overall coordination overhead remains low. The Publish-Subscribe example uses both enforceable and spontaneous transitions. Spontaneous transitions are not controlled by the JavaBIP engine which leads to low coordination overhead as illustrated in Figure 16a.

It should be noted that the complexity and the overhead induced by the JavaBIP engine depends on the size of the kernel BDD. This is characterised by the number of Boolean variables used in the encoding and their ordering in relation with the encoded constraints. Table III summarizes the number of Boolean variables used by the engine for each of the four case-studies for 5, 50 and 75 components. Notice that we take into account only enforceable ports, which correspond to transitions that are controlled by the JavaBIP engine. Table IV presents the total number of variables (which is the sum of states, ports and data variables shown in Table III) and the number of possible interactions computed by the engine for each of the four case-studies for 5, 50 and 75 components. For 75 components, the total number of variables is: 1) 521 for Camel Routes without data; 2) 669 for Camel Routes with data; 3) 645 for Trackers & Peers and 4) 226 for Publish-Subscribe.

Although the increase of the number of variables results in increasing the overhead of the engine, what really affects the performance of the engine is the complexity of the glue specification. We have used the Trackers & Peers case-study as a stress test for the JavaBIP engine. In particular, in the Trackers & Peers case-study, the number of possible interactions increases exponentially with the number of components, e.g., for 75 components, there exist $1.17 \cdot 10^{24}$ possible interactions (Table IV). Coordinating a system of such complexity with traditional techniques would be very difficult and error-prone. In JavaBIP, the full glue specification does not exceed 20 lines of code and is specified externally for the global system.

To compare the engine's execution time, we have used Camel Routes to transfer files of different size on an Intel Core i7-2640M CPU at 2.80GHz x 4 with 8GB RAM. We measured that a Camel Route needs 113ms to transfer a 3KB file, while for a 75MB file a Camel Route needs 890ms. Notice that this is "an ideal scenario" since only one Camel route is running at each time. In the case where 4 Camel Routes are running simultaneously, to transfer the 75MB file we need more than 900ms. We argue that the engine's overhead which is $<$ 1ms for 4 Camel Routes and 1 Monitor is negligible when compared to 900ms or even to 113ms. Additionally, the memory usage of the BDD Manager remains very low, less than 2MB for 75 components, for the Camel Routes with data case-study.

7. RELATED WORK

Locking provides very efficient means for coordinating concurrent access to shared resources. However, it leads to code that is hard to understand, debug and maintain. Several solutions,

DOI: 10.1002/spe

Table II. Engine times and BDD Manager peak memory usages. Time is in milliseconds and memory is in Megabytes.

Nb of comps	**Routes no data**		**Routes with data**		**Trackers & Peers**		**Publish-Subscribe**	
	Time	Memory	Time	Memory	Time	Memory	Time	Memory
5	< 1	0.010	< 1	0.015	< 1	0.640	< 1	0.026
10	< 1	0.029	< 1	0.075	2.103	2.278	< 1	0.048
15	< 1	0.047	1.147	0.099	4.264	7.584	< 1	0.084
20	< 1	0.079	1.254	0.180	6.002	10.338	< 1	0.108
25	< 1	0.099	1.585	0.220	8.980	15.932	< 1	0.130
30	1.254	0.120	1.614	0.324	12.329	23.670	< 1	0.161
35	1.328	0.169	1.895	0.456	18.643	31.896	< 1	0.184
40	1.459	0.200	2.393	0.560	24.727	43.045	< 1	0.233
45	1.874	0.238	2.731	0.700	31.187	51.598	< 1	0.251
50	2.167	0.280	3.568	0.780	38.943	69.984	< 1	0.295
55	2.346	0.340	3.796	0.840	49.766	87.097	< 1	0.315
60	2.786	0.387	5.093	0.920	63.766	99.983	< 1	0.338
65	3.286	0. 410	5.345	1.028	85.327	113.983	< 1	0.366
70	3.749	0.450	5.548	1.105	99.876	131.237	1.001	0.394
75	4.133	0.488	6.970	1.170	113.657	146.476	1.125	0.437

Table III. Number of Boolean variables used for States (S), Ports (P) and Data Variables (DV).

Nb of comps	**Routes no data**			**Routes with data**			**Trackers & Peers**			**Publish-Subscribe**		
	S	P	DV	S	P	DV	S	P	DV	S	P	DV
5	17	14	-	17	14	8	9	18	16	5	6	5
50	197	149	-	197	149	98	90	180	160	50	51	50
75	297	224	-	297	224	148	135	270	240	75	76	75

Table IV. Total number of Boolean Variables (V). Number of possible Interactions (I).

Nb of comps	**Routes no data**		**Routes with data**		**Trackers & Peers**		**Publish-Subscribe**	
	V	I	V	I	V	I	V	I
5	31	8	39	8	43	46	16	5
50	346	98	444	98	430	$2.36 \cdot 10^{16}$	151	50
75	521	148	669	148	645	$1.17 \cdot 10^{24}$	226	75

originating with the Actor model [5, 19] and the Active Object pattern [20], have been proposed for dealing with these difficulties, e.g., [21, 22, 23, 24]. (We refer the reader to [25] for an overview.) By restricting the coordination mechanisms to asynchronous message passing, the Actor model guarantees deadlock-freedom and renders the designed software easily amenable for distribution. Furthermore, it prevents components from directly accessing the state of other components, eliminating data races. However, race conditions may still occur: when two actors send messages to a third one, the resulting state of the latter depends on the order of arrival of the two messages. Although, in some cases, such race conditions may be benign, in general they might need to be avoided, representing a trade-off between performance and behavioral properties, such as predictability and reproducibility. Indeed, avoiding such race conditions requires some form of synchronization (e.g., barriers), which decreases the performance, since all synchronised processes must wait for each other. In Actor- and Active-Object-based approaches, this trade-off is realised by the use of extended features, such as synchronization on futures. Instead, the JavaBIP approach dissociates synchronization from communication of values and provides a powerful and flexible mechanism for the definition of synchronization patterns of arbitrary complexity. Thus, the above approaches focus primarily on scalability at the expense of the expressiveness of coordination primitives. They do not impose the separation of computation from coordination, thus only partially alleviating the above difficulties. Since component code substantially depends on its environment, modularity suffers. Indeed, send and receive primitives are embedded directly into the functional

DOI: 10.1002/spe

code and fairly complex message exchange protocols have to be designed, which are spread out across multiple actors and interleave with their computation code. Any modification of the coordination policy calls for corresponding modifications in the behavior of several actors. On the contrary, as illustrated by the Camel Routes example in Section 3.1, the JavaBIP approach provides high modularity, since coordination patterns can be applied or changed without requiring any modifications of existing components. Notice, finally, that asynchronous message passing used in the Actor-based approaches is encoded through the use of spontaneous transitions as illustrated by the Publish-Subscribe server example (Section 3.2); it is easy to see that synchronization on futures can also be encoded by combining spontaneous transitions with the JavaBIP synchronization mechanism.

Fractal [26] is a hierarchical component model that provides mechanisms for component control and interception. Beyond the primitive bindings between a client and server interface that commonly appear in such context, Fractal allows the so-called *composite bindings*. A composite binding is a communication path between an arbitrary number of component interfaces, assembled from binding components. Composite bindings can define multiparty interactions similar to those used in BIP and JavaBIP. The major distinction is that there is no strong restriction on the behavior of the binding components necessary to achieve that. Such use of binding components, to coordinate the execution of the functional ones, can be compared to the use of coordinators in the BIP architecture-based approach [13, 14] for enforcing property satisfaction by construction. However, enforcing properties that do not require additional state information can be achieved in both BIP and JavaBIP without introducing additional behavior, thereby providing a stronger separation of the coordination and functional concerns. Composite bindings in Fractal bear similarity to the connector architectures in SOFA 2 [27, 28], with the same remarks about separation of concerns being applicable.

The Grid Component Model (GCM) [29], is an Active-Object-based extension of Fractal for grid computing. The reference implementation of GCM, called GCM/ProActive is built on top of the ProActive [22, 30] Java middleware implementing multi-threaded active objects. The component-based approach advocated by Fractal, in general, and implemented in the GCM/ProActive framework is very close to that of BIP, particularly in their strong emphasis on the separation of concerns between coordination and computation. Apart from the above discussion of synchronization mechanisms, the main difference is that GCM/ProActive relies on code generation, as opposed to the exogenous coordination approach of JavaBIP.

It should be noted that a number of languages and frameworks mentioned above, in particular Rebeca, SOFA 2 and GCM/ProActive, have associated verification tools [23, 31, 32]. Among the behavioral model used for the verification by these tools, the one closest to BIP is the pNets [33] model used for the verification of GCM models in CADP [34]. We leave for future work a study of the connection between BIP and GCM, which would allow applying the verification tools from the BIP tool-set to GCM models. Similarly, it would be interesting to explore potential application of Rebeca or SOFA 2 verification tools to the analysis of the JavaBIP spontaneous notifications.

Xcd [35] and SIP [36], impose the separation of concerns principles while dealing with coordination of concurrent software components. Xcd makes the choice of emphasizing specifically the realizability of distributed architectures, thus excluding up-front the possibility of centralized synchronization. SIP represents the functional logic of an application as a state machine and manages synchronization contracts that specify application needs in resources. Even though this work bears similarity to our approach, it cannot be used to enforce complex synchronization scenarios as in JavaBIP.

A number of other approaches further explore the issue of coordination in concurrent environment. D [37] is a language framework based on Java and extended with aspect languages. Cool, the coordination aspect language within D, provides functionality to model mutual exclusion, synchronization state, guarded suspension and notification of threads. Coordination of multiple classes is possible, however, the synchronization of actions is hard to specify. Another aspect-oriented approach is taken in [38], where the authors focus on separating the design and the choice of a specific synchronization mechanism and present a library allowing declarative specification of synchronization using tagging. In [39], discrete-event systems theory is used to generate

DOI: 10.1002/spe

concurrency control code. Using control-flow graphs of threads coupled with predefined relevant control events, the authors build simplified FSMs for each thread. Based on these FSMs, a general supervisor is constructed, from which the concurrent code is generated using semaphores. In both of these approaches, tags or events are embedded in the code, resulting in poor separation of concerns.

In [40], a component model with explicit symbolic protocols based on Symbolic Transition Systems is presented. The authors define a component language to describe the interfaces and the protocol of a component. Connection mechanisms are represented by channels. However, only one-to-one connections are allowed. Furthermore, even though a controller is implemented whose role bears similarity to the JavaBIP executor, controller classes have to be manually created for each component in the architecture.

Automatic generation of synchronization constraints from global invariants expressed by logic formulas is discussed in [41]. The authors suggest an approach where the regions of code requiring synchronization are marked with syntactic tags and the synchronization policy for sets of dependent regions are separately defined. The limitation of this work is that it cannot define coordination scenarios taking into account variable values.

The coordination meta language for distributed objects, SynchNet, presented in [42], is inspired by the Actor model and based on Petri Nets. It allows the specification of coordination patterns for automatic generation of distributed code. However, the language uses only method labels and cannot account for data.

Using FSMs to model program behavior constitutes a common practice. In [43], the authors propose a methodology to automatically extract the finite-state models of object-oriented class interfaces. In the extracted FSMs, the states correspond to methods, while the transitions represent acceptable sequencing of method calls. The extracted models may serve as documentation, they can also be used to check whether the program exhibits expected behavior or as complementary material in a test suite. In our approach, FSMs define a relevant abstract view of the behavior of the software, which is more complex and complete than just method call sequencing and therefore can be effectively used for more complex system analysis. For instance, a state is the combination of the control location (between method calls) and data valuations, which allows, among others, to consider guards.

Similarly, in [44, 45, 46] behavioral types, represented by FSMs are used to describe the behavior of component types and their interaction protocols. The latter are constrained by the fact that behavioral types strictly describe method calls and thus, cannot describe n-ary synchronizations among components. Additionally, in contrast to JavaBIP behavioral and interaction specifications, behavioral types do not capture data transfer information. Behavioral types can be used for runtime verification through automatic generation of monitors that are executed in parallel with a system implementation. Runtime monitors, in [44], are connected to components using AspectJ. Similarly to the JavaBIP approach, this requires the developers to be aware of the APIs of the underlying components. With respect to JavaBIP, the advantage of [44] could be that components need not specifically provide spontaneous ports in their interfaces. Instead a developer could set up notifications for spontaneous transitions, as long as a sufficiently close pointcut can be defined. In JavaBIP, however, we have intentionally taken a less invasive approach that allows exogenous monitoring of components and is not subject to some of the compositionality and coupling issues linked with the aspect programming approach [47]. Similar considerations apply for AOKell [48]—an aspect-based implementation of the Fractal component model mentioned above.

In [49], the authors argue for the extension of objects with state information through Typestate-Oriented Programming. A number of works [50, 51] support the importance of this approach, which allows for the specification of classes of temporal safety properties in the form of FSMs. This extension of types serves in checking whether an operation is allowed in a specific context. These works address the problems of retrieving or verifying method call sequences, whereas JavaBIP focuses on enforcing coordination.

The Behavior Driven Development [52] software approach puts a strong focus on explicitly stating the behavior of system components. In particular, Behavior Driven Development focuses on understanding the behavior of the components defined as simple *given-when-then* scenarios.

DOI: 10.1002/spe

Such a scenario can be simply explained as follows: *given* the state of the component, *when* an action is executed the component reaches the corresponding *then* state. The JBehave tool [53] that supports Behavior Driven Development also requires a user to create an object model (an instance of a Java class) that directly maps scenarios to functions within this object model. Similarly to Behavior Driven Development, our approach puts a strong focus on explicitly stating the behavior of the system components. The JavaBIP mechanism for behavior specification could be used to define given-when-then scenarios for all transitions within a Behavior specification.

Another example is the Domain Driven Design (DDD) [54]. In DDD, it is important to model a system using a ubiquitous language based on the business domain, so the business vocabulary permeates the code base. JavaBIP specifications allow creating an abstract view of a software entity where one can create a clean model of the system component using terms from the language of the business domain. At the same time, BIP specification may internally interact with existing software components originating from legacy systems that have different vocabulary as well as often technical (non-business) nature. Business transitions specified within a Behavior specification can internally use functionality of technical components, e.g., transactional support within a database. A *Bounded context* concept from DDD encourages explicitly stating the boundaries for the application of the models. By encapsulating technical components, JavaBIP Behavior specifications can provide abstract representations of their behaviour at levels of detail appropriate for the corresponding business domains.

8. CONCLUSION AND FUTURE WORK

The main goal of our work was to allow an exogenous approach to the coordination of software components relying, for the interaction with the controlled components, on existing APIs. To this end, JavaBIP specifications are defined as separate Java classes that, on one hand, allow the interaction with *other* components through the mediation of the JavaBIP engine and, on the other hand, interact with the *controlled* components through the provided APIs and notifications of spontaneous events. All JavaBIP models are defined separately from the functional code of the components. The behavior specifications are very well localized, which makes them easy to provide. Furthermore, the glue and data-wire specifications can be defined and modified without altering the functional code, which is impossible with traditional approaches.

We have presented experimental results that validate the effectiveness of JavaBIP for the coordination of software components. Verification tools, e.g., DFinder and nuXmv, can be used to validate JavaBIP specifications, ensuring correctness of the designed systems. JavaBIP coordination has been tested and validated in Connectivity Factory™ by Crossing-Tech S.A.

In a somewhat broader context, we argue that the existing notion of an interface is not sufficient for ensuring correct manipulation of objects. Indeed, Java interfaces only specify the methods that must be implemented by a class, but the information about *when* these methods can be called is hidden from other components: the responsibility of checking whether the state of an object allows the execution of a given method is relegated to the method itself. This has several important consequences: 1) all the information about the conditions when the method can be called must be available at the design time; 2) it is difficult to impose additional restrictions on these conditions without having to change the existing implementation; 3) this information cannot be used for coordinating the execution of other components. Our work addresses all these issues by extending Java interfaces with state information. This is a first step in the direction that we consider important in the future development of programming languages such as Java. Notice that a class can implement several interfaces. Similarly, several FSMs can represent independent abstract views of different aspects of the component behavior. Extending interfaces with state information is compatible with interface composition and inheritance.

Future work comprises several directions: extending our framework with additional coordinators, e.g., to manage priorities, dynamic component creation and resource allocation policies; incorporating run-time state validity verification by annotating states with assertions that would

DOI: 10.1002/spe

be checked after firing a transition; finally, improving scalability by combining existing techniques from the BIP framework for parallelizing and distributing the JavaBIP engine.

Currently, in JavaBIP, we are using the Require and Accept macros to specify the glue. These macros are based on first-order interaction logic and define architectural constraints [13]. In order to generalize these specifications, we plan to use configuration logics [55, 56] and architecture diagrams [15, 57] to define not a single architecture but families of architectures that share common characteristics such as the type of the involved components and the properties they impose. By using families of architectures, we facilitate reusability and correctness-by-construction. A software developer that uses JavaBIP would not always need to create new glue specifications but rather generate them directly from a library of specifications, which have been previously checked for correctness.

ACKNOWLEDGEMENTS

Our implementation of the Publish-Subscribe example in JavaBIP was based on the native Java code by Thomas Ropars and Martin Biely for the Concurrency class taught at EPFL by André Schiper. We are very grateful to the anonymous reviewers for their comments and constructive suggestions that have made significant contributions to improving this paper. This work was partially supported by the Swiss Commission for Technology and Innovation (CTI 14432.1 PFES-ES).

REFERENCES

1. The Apache Software Foundation. Apache Camel:Routes. http://camel.apache.org/routes.html. (Accessed on 12/02/2016.).
2. Williams A. *C++ Concurrency in Action: Practical Multithreading*. 1st edn., Manning Publications: Greenwich, CT, USA, 2012.
3. Lea D. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.
4. Basu A, Bensalem S, Bozga M, Combaz J, Jaber M, Nguyen TH, Sifakis J. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software* 2011; **28**(3):41–48, doi:10.1109/MS.2011.27.
5. Agha G. *Actors: a model of concurrent computation in distributed systems*. MIT Press: Cambridge, MA, USA, 1986.
6. Bensalem S, Bozga M, Nguyen TH, Sifakis J. D-Finder: A tool for compositional deadlock detection and verification. *Computer Aided Verification*, *Lecture Notes in Computer Science*, vol. 5643, Springer, Springer Berlin Heidelberg: Grenoble, France, 2009; 614–619.
7. Bliudze S, Cimatti A, Jaber M, Mover S, Roveri M, Saab W, Wang Q. Formal verification of infinite-state BIP models. *13th International Symposium on Automated Technology for Verification and Analysis*, *LNCS*, vol. 9364, Springer International Publishing: Shanghai, China, 2015; 326–343, doi:10.1007/978-3-319-24953-7_25.
8. Bliudze S, Sifakis J. The algebra of connectors—structuring interaction in BIP. *IEEE Transactions on Computers* 2008; **57**(10):1315–1330, doi:http://doi.ieeecomputersociety.org/10.1109/TC.2008.26.
9. Bliudze S, Sifakis J, Bozga MD, Jaber M. Architecture internalisation in BIP. *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering (CBSE '14)*, ACM: Marcq-en-Bareul, France, 2014; 169–178, doi:10.1145/2602458.2602477.
10. Bliudze S, Mavridou A, Szymanek R, Zolotukhina A. Coordination of software components with BIP: Application to OSGi. *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, MiSE 2014, ACM: New York, NY, USA, 2014; 25–30, doi:10.1145/2593770.2593777.
11. Bozga M, Jaber M, Maris N, Sifakis J. Modelling dynamic architectures using Dy-BIP. *Lecture Notes in Computer Science* 2012; **7306**:1–16, doi:10.1007/978-3-642-30564-1_1.
12. Bliudze S, Sifakis J. Synthesizing glue operators from glue constraints for the construction of component-based systems. *Software Composition*, Apel S, Jackson E (eds.), Lecture Notes in Computer Science, Springer: Zurich, Switzerland, 2011; 51–67, doi:10.1007/978-3-642-22045-6_4.
13. Attie P, Baranov E, Bliudze S, Jaber M, Sifakis J. A general framework for architecture composability. *12th International Conference on Software Engineering and Formal Methods (SEFM 2014)*, Giannakopoulou D, Salaün G (eds.), no. 8702 in LNCS, Springer International Publishing: Switzerland, 2014; 128–143.
14. Attie P, Baranov E, Bliudze S, Jaber M, Sifakis J. A general framework for architecture composability. *Formal Aspects of Computing* Apr 2016; **18**(2):207–231.
15. Mavridou A, Stachtiari E, Bliudze S, Ivanov A, Katsaros P, Sifakis J. Architecture-based design: A satellite on-board software case study. *13th International Conference on Formal Aspects of Component Software (FACS 2016)*, 2016. To appear.
16. Harel D. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* Jun 1987; **8**(3):231–274, doi:10.1016/0167-6423(87)90035-9.
17. Bliudze S, Sifakis J. Causal semantics for the algebra of connectors. *Formal Methods in System Design* 2010; **36**(2):167–194, doi:10.1007/s10703-010-0091-z.
18. Akers S. Binary decision diagrams. *IEEE Transactions on Computers* 1978; **C-27**(6):509–516, doi:10.1109/TC.1978.1675141.

19. Gupta M. *Akka Essentials*. Community experience distilled, Packt Publishing, 2012.
20. Lavender RG, Schmidt DC. Active object: An object behavioral pattern for concurrent programming. *Pattern Languages of Program Design*, vol. 2, Vlissides JM, Coplien JO, Kerth NL (eds.). Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1996; 483–499. URL http://dl.acm.org/citation.cfm?id=231958.232967.
21. Johnsen EB, Hähnle R, Schäfer J, Schlatte R, Steffen M. ABS: A core language for abstract behavioral specification. *Revised Papers of the 9th International Symposium on Formal Methods for Components and Objects , FMCO 2010*, *Lecture Notes in Computer Science*, vol. 6957, Springer: Graz, Austria, 2010; 142–164, doi:10.1007/978-3-642-25271-6_8. URL http://dx.doi.org/10.1007/978-3-642-25271-6_8.
22. Caromel D, Henrio L, Serpette BP. Asynchronous sequential processes. *Information and Computation* 2009; **207**(4):459–495, doi:10.1016/j.ic.2008.12.004. URL http://dx.doi.org/10.1016/j.ic.2008.12.004.
23. Sirjani M, Movaghar A, Shali A, De Boer FS. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informaticae* 2004; **63**(4):385–410.
24. Cavé V, Budimlić Z, Sarkar V. Comparing the usability of library vs. language approaches to task parallelism. *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, ACM: Reno, Nevada, USA, 2010; 9:1–9:6, doi:10.1145/1937117.1937126.
25. Henrio L, Rochas J. From modelling to systematic deployment of distributed active objects. *Proceedings of the 18^{th} IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION 2016)*, *Lecture Notes in Computer Science*, vol. 9686, Springer: Heraklion, Greece, 2016; 208–226, doi:10.1007/978-3-319-39519-7_13. URL http://dx.doi.org/10.1007/978-3-319-39519-7_13.
26. Bruneton E, Coupaye T, Leclercq M, Quéma V, Stefani JB. The Fractal component model and its support in Java. *Software: Practice and Experience* 2006; **36**(11–12):1257–1284, doi:10.1002/spe.767.
27. Bures T, Plasil F. Communication style driven connector configurations. *Proceedings of the 1^{st} International Conference on Software Engineering Research and Applications (SERA 2003)*, *Lecture Notes in Computer Science*, vol. 3026, Springer Berlin Heidelberg: San Francisco, CA, USA, 2004; 102–116, doi:10.1007/978-3-540-24675-6_11. URL http://dx.doi.org/10.1007/978-3-540-24675-6_11.
28. Bures T, Hnetynka P, Plasil F. Sofa 2.0: Balancing advanced features in a hierarchical component model. *Proceedings of the 4^{th} Int. Conf. on Software Engineering Research, Management and Applications (SERA 2006)*, IEEE Computer Society: Seattle, Washington, USA, 2006; 40–48, doi:10.1109/SERA.2006.62.
29. Baude F, Caromel D, Dalmasso C, Danelutto M, Getov V, Henrio L, Pérez C. GCM: A grid extension to Fractal for autonomous distributed components. *annals of telecommunications - annales des télécommunications* 2009; **64**:5–24, doi:10.1007/s12243-008-0068-8.
30. ProActive Parallel Suite. Available online: http://proactive.activeeon.com/. Accessed on 17/10/2016.
31. Kofron J. Checking software component behavior using behavior protocols and spin. *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, ACM: New York, NY, USA, 2007; 1513–1517, doi:10.1145/1244002.1244326. URL http://doi.acm.org/10.1145/1244002.1244326.
32. Henrio L, Kulankhina O, Li S, Madelaine E. Integrated environment for verifying and running distributed components. *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE 2016)*, *Lecture Notes in Computer Science*, vol. 9633, Stevens P, Wąsowski A (eds.), Springer Berlin Heidelberg, 2016; 66–83, doi:10.1007/978-3-662-49665-7_5. URL http://dx.doi.org/10.1007/978-3-662-49665-7_5.
33. Ameur-Boulifa R, Henrio L, Madelaine E, Savu A. Behavioural semantics for asynchronous components. *Research Report RR-8167*, INRIA Dec 2012. URL https://hal.inria.fr/hal-00761073.
34. Garavel H, Lang F, Mateescu R, Serwe W. CADP 2010: A toolbox for the construction and analysis of distributed processes. *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, *Lecture Notes in Computer Science*, vol. 6605, Springer Berlin Heidelberg: Saarbrücken, Germany, 2011; 372–387, doi:10.1007/978-3-642-19835-9_33. URL http://dx.doi.org/10.1007/978-3-642-19835-9_33.
35. Ozkaya M, Kloukinas C. Design-by-contract for reusable components and realizable architectures. *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, ACM: New York, NY, USA, 2014; 129–138, doi:10.1145/2602458.2602463.
36. Huang Y, Cheung E, Dillon LK, Stirewalt REK. A thread synchronization model for SIP servlet containers. *IPTComm '09*, ACM, 2009; 7:1–7:12, doi:10.1145/1595637.1595647.
37. Lopes CIV. D: A language framework for distributed programming. PhD Thesis, Northeastern University 1997.
38. Zhang C. FlexSync: An aspect-oriented approach to Java synchronization. *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, IEEE Computer Society: Washington, DC, USA, 2009; 375–385, doi:10.1109/ICSE.2009.5070537.
39. Dragert C, Dingel J, Rudie K. Generation of concurrency control code using discrete-event systems theory. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, ACM: New York, NY, USA, 2008; 146–157, doi:10.1145/1453101.1453122.
40. Pavel S, Noyé J, Poizat P, Royer JC. A Java implementation of a component model with explicit symbolic protocols. *Software Composition*, *LNCS*, vol. 3628. Springer, 2005; 115–124, doi:10.1007/11550679_9.
41. Deng X, Dwyer MB, Hatcliff J, Mizuno M. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, ACM: New York, NY, USA, 2002; 442–452, doi:10.1145/581339.581394.
42. Ziaei R, Agha G. Synchnet: A petri net based coordination language for distributed objects. *Proceedings of the 2Nd International Conference on Generative Programming and Component Engineering*, GPCE '03, Springer-Verlag New York, Inc.: New York, NY, USA, 2003; 324–343.

43. Whaley J, Martin MC, Lam MS. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes* Jul 2002; **27**(4):218–228, doi:10.1145/566171.566212.
44. *Behavioral Types for Component-Based Development of Cyber-Physical Systems*, *Lecture Notes in Computer Science*, vol. 9509, Springer: York, UK, 2015.
45. Blech JO. Towards a framework for behavioral specifications of OSGi components. *Formal Engineering Approaches to Software Components and Architectures*, *Electronic Proceedings in Theoretical Computer Science*, vol. 108, Rome, Italy, 2013; 79–93.
46. Blech JO. Ensuring OSGi component based properties at runtime with behavioral types. *10th Workshop on Model-Driven Engineering, Verification, and Validation*, CEUR Workshop Proceedings: Miami, Florida, USA, 2013; 51–60.
47. Gybels K, Brichau J. Arranging language features for more robust pattern-based crosscuts. *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, AOSD '03, ACM: New York, NY, USA, 2003; 60–69, doi:10.1145/643603.643610. URL http://doi.acm.org/10.1145/643603.643610.
48. Seinturier L, Pessemier N, Duchien L, Coupaye T. A component model engineered with components and aspects. *Proceedings of the 9th International Symposium on Component-Based Software Engineering, CBSE 2006*, *Lecture Notes in Computer Science*, vol. 4063, Springer, 2006; 139–153, doi:10.1007/11783565_10. URL http://dx.doi.org/10.1007/11783565_10.
49. Aldrich J, Sunshine J, Saini D, Sparks Z. Typestate-oriented programming. *OOPSLA '09*, ACM: New York, NY, USA, 2009; 1015–1022, doi:10.1145/1639950.1640073.
50. DeLine R, Fähndrich M. Typestates for objects. *ECOOP 2004*, *LNCS*, vol. 3086. Springer, 2004; 465–490, doi:10.1007/978-3-540-24851-4_21.
51. Joshi P, Sen K. Predictive typestate checking of multithreaded Java programs. *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, IEEE: L'Aquila, Italy, 2008; 288–296, doi:10.1109/ASE.2008.39.
52. Solís C, Wang X. A study of the characteristics of Behaviour Driven Development. *37th EUROMICRO Conference on Software Engineering and Advanced Applications*, IEEE, 2011; 383–387, doi:10.1109/SEAA.2011.76.
53. North D. JBehave. A framework for behaviour driven development (BDD) 2011.
54. Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.
55. Mavridou A, Baranov E, Bliudze S, Sifakis J. Configuration Logics: Modelling architecture styles. *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, *Lecture Notes in Computer Science*, vol. 9539, Braga C, Ölveczky PC (eds.), Springer, 2015; 256–274, doi:10.1007/978-3-319-28934-2_14. URL http://dx.doi.org/10.1007/978-3-319-28934-2_14.
56. Mavridou A, Baranov E, Bliudze S, Sifakis J. Configuration logics: Modeling architecture styles. *Journal of Logical and Algebraic Methods in Programming* 2017; **86**(1):2–29, doi:10.1016/j.jlamp.2016.05.002.
57. Mavridou A, Baranov E, Bliudze S, Sifakis J. Architecture diagrams: A graphical language for architecture style specification. *Proceedings 9th Interaction and Concurrency Experience (ICE)*, *Electronic Proceedings in Theoretical Computer Science*, vol. 223, 2016; 83–97, doi:10.4204/EPTCS.223.6.