Coordination of Dynamic Software Components with JavaBIP

Anastasia Mavridou[1(✉)], Valentin Rutz[2], and Simon Bliudze[3]

[1] Institute for Software Integrated Systems, Vanderbilt University, Nashville, USA
anastasia.mavridou@vanderbilt.edu
[2] Microsoft Corporation, Dublin, Ireland
[3] Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Abstract. JavaBIP allows the coordination of software components by clearly separating the functional and coordination aspects of the system behavior. JavaBIP implements the principles of the BIP component framework rooted in rigorous operational semantics. Recent work both on BIP and JavaBIP allows the coordination of static components defined prior to system deployment, i.e., the architecture of the coordinated system is fixed in terms of its component instances. Nevertheless, modern systems, often make use of components that can register and deregister dynamically during system execution. In this paper, we present an extension of JavaBIP that can handle this type of dynamicity. We use first-order interaction logic to define synchronization constraints based on component types. Additionally, we use directed graphs with edge coloring to model dependencies among components that determine the validity of an online system. We present the software architecture of our implementation, provide and discuss performance evaluation results.

1 Introduction

We have previously introduced JavaBIP [9,10] that allows coordinating software components exogenously, i.e., without requiring access to component source code. JavaBIP relies on the following observations. Domain specific components have states (e.g., idle, working) 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 a component changes its state spontaneously. Thus, component behavior can be easily represented by Finite State Machines (FSMs).

JavaBIP brings the BIP principles into a more general software engineering context than that of embedded systems, in which code generation might not be desirable due to continuous code updates. Thus, to use JavaBIP, instead of generating Java code from the BIP modeling language, developers must provide—for the relevant components—the corresponding FSMs in the form of annotated Java classes. The FSMs describe the protocol that must be respected to access a shared resource or use a service provided by a component. FSM transitions

© Springer International Publishing AG 2017
J. Proença and M. Lumpe (Eds.): FACS 2017, LNCS 10487, pp. 39–57, 2017.
DOI: 10.1007/978-3-319-68034-7_3

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.

For component coordination, JavaBIP provides two primitive mechanisms: (1) multi-party synchronizations of component transitions and (2) asynchronous event notifications. The latter embodies the reactive programming paradigm. In particular, JavaBIP extends the Actor model [1], since event notifications can be used to emulate asynchronous messages, while providing the synchronization of component transitions as a primitive mechanism gives developers a powerful and flexible tool to manage coordination. The synchronization of component transitions is managed by a runtime called JavaBIPEngine, which, for simplicity, we call "engine" in the rest of the paper. Notice that in a completely asynchronous system the engine is not needed.

JavaBIP clearly separates system-wide coordination policies from component behavior. Synchronization constraints, defining the possible synchronizations among transitions of different components i.e., the set of possible component *interactions*, are specified independently from the design of individual components in dedicated XML files. This separation of functional and coordination aspects greatly reduces the burden of system complexity. Finally, integration with the BIP framework, through a JavaBIP-to-BIP code generation tool, allows the use of existing deadlock-detection and model checking tools [7,8] ensuring the correctness of JavaBIP systems.

The previous implementation of JavaBIP [10] was static. To coordinate a system, the full set of components had be registered before starting the engine. No components could be added on-the-fly and, most importantly, if a failure occurred in a single component, the engine execution had to stop and the full set of constraints had to be computed anew. Notice that none of the current BIP implementations [5,6,11] allows to add or remove components on-the-fly, including DyBIP presented in [12] that allows dynamically changing the set of interactions among a fixed set of components at runtime. This might be problematic, since modern systems, e.g., large banking systems or modular smartphones, make use of components that can register and deregister during system execution.

To allow dynamicity in JavaBIP, we use first-order interaction logic to describe synchronization constraints on component types. As a result, a developer can write synchronization constraints without knowing the exact number of components in the system. Thus, component instances of known types, i.e., types for which synchronization constraints exist, can register at runtime without any additional input from the developer. To optimize JavaBIP performance, we have introduced a notion of system validity: *a system is valid if and only if its set of possible interactions is not empty.* The notion of validity allows the engine to be started and stoped automatically at runtime by just checking the status of the system. By stopping the engine if the system is invalid, we eliminate any processing time needed by the engine. To check system validity, we use directed graphs with edge coloring to model component synchronization dependencies.

Notice that the introduced notion of validity is only relevant for the engine: in an invalid system components can still communicate asynchronously.

We have extended the interface and implementation of the engine to register, deregister, and pause a component at runtime. The difference between pausing and deregistering a component is as follows. If a component deregisters, then the engine clears all the associated data and references to this component; other components cannot synchronize with the deregistered component unless it registers anew. If a component is paused, other components cannot synchronize with it but the engine keeps all associated data and references to it; the paused component can start synchronizing with other components by simply informing the engine that it is back on track.

The rest of the paper is structured as follows. Section 2 describes our motivating case study. Section 3 presents the JavaBIP framework and the macro-notation used to specify JavaBIP synchronization constraints on component types. Section 4 presents the notion of JavaBIP system validity and the construction of validity graphs. Section 5 describes the implemented software architecture and presents performance results. Section 6 discusses related work. Section 7 summarizes the results and future work directions.

2 Motivating Case Study

Modular phones require application layer specifications that can handle dynamic device insertion and removal at runtime. In the rest of the paper, we refer to the phone's devices as *modules*. In this case study, we model in JavaBIP some of the application layer protocols offered by Google's Greybus specification[1].

Figure 1 illustrates the composite component types, of the case study. Greybus requires that exactly one application processor (AP) is present in the system

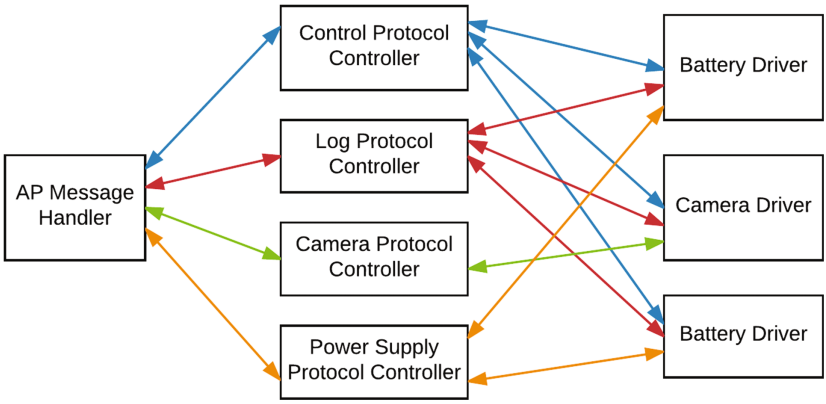

Fig. 1. Modular phone architecture.

[1] https://github.com/projectara/greybus-spec.

```
Application Layer ::= (AP Message Handler).(Controller)+.(Driver)*
AP Message Handler ::= (AP Request Worker).(AP Response Worker).
                        (AP Message Worker).(AP Receiver Fifo)
Controller ::= (Control Protocol Controller).(Log Protocol Controller).
          (Camer Protocol Controller).(Power Supply Protocol Controller)
Driver ::= (Battery Driver)*.(Camera Driver)*
Camera Driver ::= (Control Connect Handler).(Control Disconnect Handler).
             (Log Handler).(Camera Capture Handler).(Camera Stream Handler)
Battery Driver ::= (Control Connect Handler).(Control Disconnect Handler).
                                    (Log Handler).(Power Supply Handler)
```

Fig. 2. Hierarchical decomposition of the `Application Layer` into components.

for storing user data and executing applications. We consider two types of modules that can be inserted on the phone's frame at runtime: (1) power supply modules, e.g., batteries and (2) cameras. Any number of instances of these two types can be inserted or removed from the phone at runtime. Figure 1 presents an example configuration of a phone, in which two battery and one camera modules are connected. These modules communicate with the AP through dedicated device class connection protocols: the *camera*, *power supply*, and *log* protocols. The latter can be used by any module to send human-readable debug log messages to AP. Additionally, AP uses the *control protocol* to perform basic initialization and configuration actions with other modules. If no power supply or camera modules are connected, the system configuration would consist of the `AP Message Handler`, `Control Protocol Controller`, the `Log Protocol Controller`, `Camera Protocol Controller`, and the `Power Supply Protocol Controller` composite components. The grammar in Fig. 2 shows how to obtain the desired systems as the incremental composition of components. Operators . (dot), $\cdot^*$ and $\cdot^+$ are used as usual to denote composition and repetition. Notice that Fig. 1 illustrates only one of the possible system configurations that are described by the grammar in Fig. 2. A detailed description of the system's componentization and interaction model can be found in [31].

A Greybus protocol defines a number of Greybus operations, which are *request-response pairs* of remote procedure calls from one module to another. The bi-directional arrows in Fig. 1 represent Greybus operations. For instance, the AP very often needs to retrieve information from other modules. This requires that a message requesting information be paired with a response message containing the information requested. In many cases, Greybus operations need to be performed in a specific order. Additionally, the access to shared resources such as memory and logging services needs to be controlled among modules. We enforce action flow of Greybus operations, as well as controlled access to the phone's shared resources with JavaBIP. We developed the case study using the WebGME-BIP design studio[2], the complete system exceeds 2000 lines of code.

[2] https://github.com/anmavrid/webgme-bip.

3 The JavaBIP Component Framework

JavaBIP implements the BIP (Behavior-Interaction-Priority) coordination mechanism [5], for coordination of concurrent components. In BIP, the behavior of components is described by Finite State Machines (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. Component coordination is defined in BIP by means of *interaction models*, i.e., sets of interactions. Interactions are sets of ports that define allowed synchronizations among components.

JavaBIP takes as input the *system specification*, which is provided by the user and consists of the following:

- A *behavior specification* for each component type, which is an FSM extended with ports and data provided as an annotated Java class.
- The *glue specification*, which is the interaction model of the system, is provided as an XML file. It specifies how the transitions of different component types must be synchronized, i.e., synchronization constrains.
- The optional *data-wire specification*, which is the data transfer model of the system, is provided as an XML file. It specifies which and how data are exchanged among component types.

For property analysis, the system specification can be automatically translated into an equivalent model of the system in the BIP language. This model can then be verified for deadlock freedom or other properties, using DFinder [7], ESST or nuXmv [8]. Other analyses can be performed using any tool for which a model transformation from BIP is available.

3.1 Glue Specification

The glue specification is defined in JavaBIP through a macro-notation, similar to the one introduced in [12], based on first-order interaction logic. This notation imposes synchronization constraints based on component types rather than on component instances, which allows a developer to write a glue specification without knowing the exact number of components in the system. Instances of component types for which synchronization constraints exist in the glue specification can be dynamically registered or deregistered at runtime without requiring additional input or changes in the glue specification. We briefly present the JavaBIP Require/Accept macro-notation used for the glue specification. The JavaBIP Require/Accept macro-notation was previously presented in [10]. We also refer the interested reader to [31], where we present in detail the propositional interaction logic, the first-order extension and the Require/Accept macro-notation.

Consider a port p of a component type T, which labels one or more transitions of T. The associated synchronization constraint to all transitions of T labeled by p is the conjunction of two constraints: the *causal* and *acceptance* constraints.

Two macros are used: (1) the **Require** macro and (2) the **Accept** macro to define the causal and acceptance constraints, respectively. Next, we describe the meaning of the two macros through representative examples. The generalization of the above definitions to more complex macros is straightforward, but cumbersome. Therefore we omit it here.

The Require Macro is used to specify ports required for synchronization. Let $T^1, T^2 \in \mathcal{T}$ be two component types. The macro

$$T_1.p \;\; \textbf{Require} \;\; T_2.q$$

means that, to participate in an interaction, each of the ports p of component instances of type T_1 requires synchronization with *precisely one* of the ports q of component instances of type T_2. Notice that the cardinality of required component instances is explicit: should two instances of the same port type be required, this is specified by explicitly putting the required port type twice, e.g.,

$$T_1.p \;\; \textbf{Require} \;\; T_2.q \; T_2.q \,,$$

and so on for higher cardinalities. We call *effect* what is specified in the left-hand side of **Require** (e.g., $T_1.p$) and *cause* what is specified in the right-hand side (e.g., $T_2.q \; T_2.q$). A cause consists of a set of *OR-causes*, where each OR-cause is a set of ports. For p to participate in an interaction, all the ports belonging to at least one of the OR-causes must synchronize. For instance,

$$T_1.p \;\; \textbf{Require} \;\; T_2.q \; T_2.q \; ; \; T_2.r$$

means that p requires either the synchronization of two instances of q or one instance of r. Notice the semicolon that separates the two OR-causes.

The Accept Macro defines optional ports for synchronization, i.e., it defines the boundary of interactions. This is expressed by explicitly excluding from interactions all the ports that are not accepted. Let $T^1, T^2 \in \mathcal{T}$ be two component types. The following:

$$T_1.p \;\; \textbf{Accept} \;\; T_2.q$$

means that p accepts the synchronization of instances of q but does not accept instances of any other port types.

4 Defining System Validity

In the previous, static JavaBIP implementation, a developer had to first register all components to the engine and then start the engine manually. Since, in the presented implementation, components may register or deregister on-the-fly, we introduce a notion of validity so that depending on whether there are enough registered components, the engine can automatically start or stop its execution. We start by formally defining components, BIP systems and valid BIP systems.

Definition 1 (BIP Component). *A BIP component B is an FSM represented by a triple $(Q, P, \rightarrow)$, where Q is a set of states, P is a set of communication ports, $\rightarrow \subseteq Q \times P \times Q$ is a set of transitions, each labeled by a port.*

Below, we use the common notation, writing $q \xrightarrow{p} q'$ instead of $(q, p, q') \in \rightarrow$.

Definition 2 (BIP System). *A BIP system is defined by a composition operator parameterized by a set of interactions $\gamma \subseteq 2^P$. $\mathcal{B}_n = \gamma(B_1, \ldots, B_n)$ is an FSM represented by a triple $(Q, \gamma, \rightarrow)$, where $Q = \prod_{i=1}^{n} Q_i$ and $\rightarrow$ is the least set of transitions satisfying the following rule:*

$$\frac{a = \{p_i\}_{i \in I} \in \gamma \qquad \forall i \in I : q_i \xrightarrow{p_i} q'_i \qquad \forall i \notin I : q_i = q'_i}{(q_1, \ldots, q_n) \xrightarrow{a} (q'_1, \ldots, q'_n)}$$

The inference rule says that a BIP system, consisting of n components, can execute an interaction $a \in \gamma$, iff for each port $p_i \in a$, the corresponding component B_i, can execute a transition labeled with p_i; the states of components that do not participate in the interaction remain the same. The set of possible interactions of a BIP system is defined in JavaBIP by the glue specification, i.e., the set of Require and Accept macros. We write $B{:}T$ to denote a component B of type T. We denote by $\mathcal{T}$ the set of all component types of a BIP system.

Definition 3 extends Definition 2 to describe a valid BIP system. System validity is defined from the perspective of starting/stopping the engine execution. Notice that even if a system is not valid according to Definition 3, JavaBIP components can communicate in an asynchronous manner.

Definition 3 (Valid BIP System). *A BIP system $(Q, \gamma, \rightarrow)$ is valid iff $\gamma \neq \emptyset$.*

Remark 1. In Definitions 1 and 2, for the sake of simplicity, we omit the presentation of data-related aspects. However, it should be noted that the full JavaBIP [10] allows data variables within components. In such cases, component transitions can be guarded by Boolean predicates on data variables. Notice that in Definition 3 we do not consider guards. This is a design choice that we made. The result of guard evaluation might easily change multiple times throughout the system lifecycle, e.g., based on the components internal state or on component interaction. Thus, it is undesirable to base engine execution on such often recurring changes, which could actually result in increasing the engine's overhead.

Definition 3 says that a BIP system is valid if and only if there are enough registered components such that the interaction set of the system is not empty. To determine the validity of a system, we use directed graphs with edge coloring to model dependencies among components. The generation of the validity graph is based on the Require macros of the glue specification, since these define the minimum number of required interactions among the components. The complete glue specification is used by the engine for orchestrating component execution.

Definition 4 (Validity graph). *A labelled graph $G = (\mathcal{T}, E, c)$ is the validity graph of a set of Require macros iff:*

1. *the vertex set $\mathcal{T}$ is the set of component types defined in the Require macros;*
2. *the edge set E contains a directed edge (T_1, T_2) iff there exists a Require macro that contains T_1 in the effect and T_2 in an OR-cause;*
3. *for each edge $(T_1, T_2) \in E$, the counter $c : E \rightarrow \mathbb{Z}$ is initialized with the cardinality of T_2 in the corresponding OR-cause.*

The edges of the graph are colored such that: 1) all edges corresponding to an OR-cause of a Require macro are colored the same; 2) edges corresponding to different OR-causes are colored differently.

Clearly, there always exists a validity graph for any set of Require macros. Note that the outgoing edges of two different vertices may have the same color.

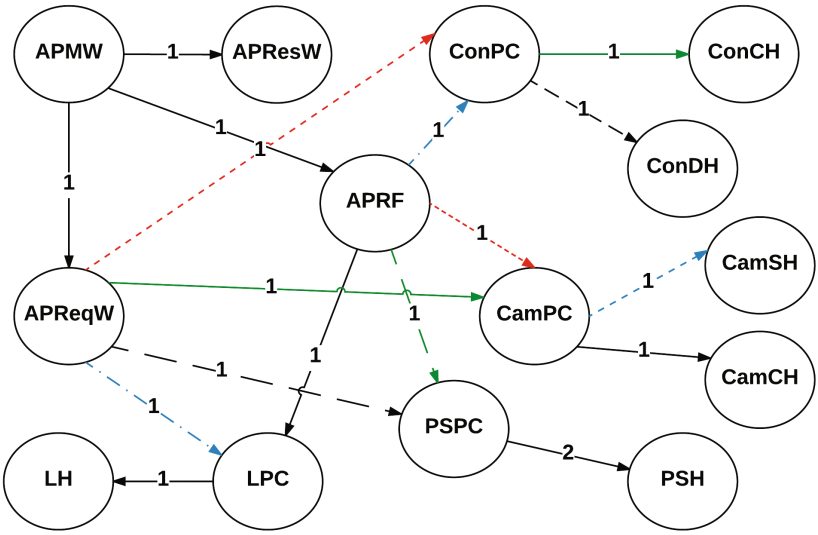

Fig. 3. Validity graph of the Modular Phone case study.

Figure 3 shows part of the validity graph of the case study (the full graph can be found in [31]). There are 13 vertices, each one representing an atomic component type from Fig. 2. Due to space limitations, we substituted the full names with their acronyms. For instance, we substituted `AP Message Handler` by `APMW`. In case of acronym conflicts, we added more letters, e.g., we substituted `AP Request Worker` by `APReqW`, and `AP Response Worker` by `APResW`.

Let us now consider two of the Require macros of the case study (the full set of Require and Accept macros can be found in [31]).

`PSPC.snd` **Require** `PSH.rcv PSH.rcv`
`APRF.snd` **Require** `ConPC.rvc; CamPC.rcv; PSPC.rcv; LPC.rcv`

Since, component type `Power Supply Protocol Controller` requires synchronization with two instances of component type `Power Supply Handler`, there is an edge from vertex `PSPC` to vertex `PSH` labeled by a counter initialized to 2. Furthermore, component type `AP Receiver Fifo` requires synchronization either with an instance of component type `Control Protocol Controller` or an instance of component type `Camera Protocol Controller` or an instance of `Power Supply Protocol Controller` or an instance of `Logger Protocol Controller`. Thus, there are four outgoing edges from vertex `APRF`, each labeled by a counter initialized to 1 and colored by a different color, to vertices `ConPC`, `CamPC`, `PSPC`, and `LPC`, respectively. In Fig. 3, edges with different colors are also represented by different line styles.

Definition 5 (Dynamic change in validity graph). *In the event of a dynamic change, a validity graph is updated as follows:*

1. *If a component instance of type* `T` *is registered, the counters of all incoming edges of vertex* `T` *are decremented by* 1.
2. *If a component instance of type* `T` *is deregistered or paused, the counters of all incoming edges of vertex* `T` *are incremented by* 1.

Proposition 1 (Determining system validity). *Consider a BIP system and a corresponding validity graph. The BIP system is valid iff for at least one vertex of the validity graph, an instance of the vertex's corresponding type is registered and the counters of all outgoing edges of at least one color are equal to or less than* 0.

Example 1. Figure 4 presents the changes in the validity graph of Fig. 3[3], after the registration of an instance of each of the `APMW`, `APResW`, `APRF`, and `PSH` component types. System validity will be guaranteed if at least one instance of a component type contained in one of the red dashed boxes gets registered.

To start and stop the engine, we determine first whether the system is valid by using Proposition 1. Nevertheless, we do not need to check system validity every time a component registers/deregisters/pauses. Corollaries 1 and 2 define such cases. The proofs of Proposition 1 and Corollaries 1 and 2 can be found in [31].

Corollary 1. *If a BIP system $\mathcal{B}_n$ is valid and a component is registered, then the new BIP system $\mathcal{B}_{n+1}$ is also valid.*

Corollary 2. *If a BIP system $\mathcal{B}_n$ is invalid and a component is deregistered or paused, then the new BIP system $\mathcal{B}_{n-1}$ is also invalid.*

[3] The complete validity graph of the case study can be found in [31].

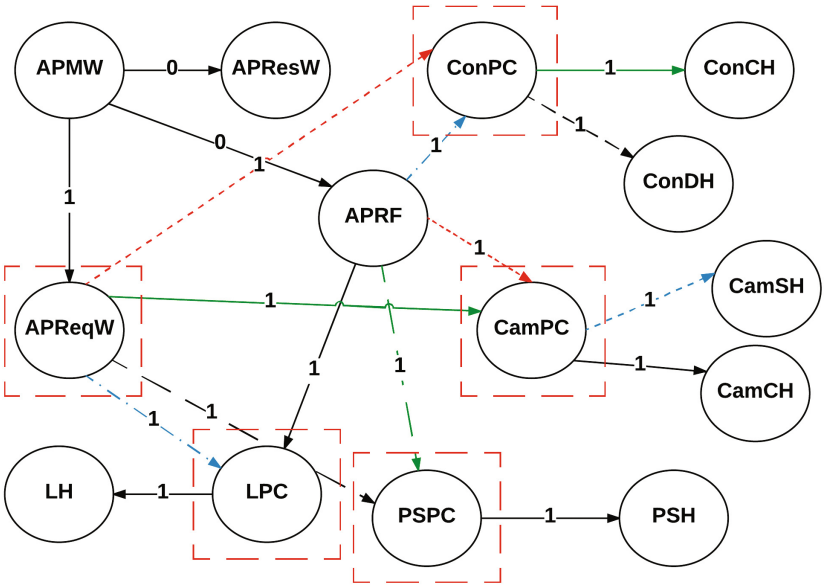

Fig. 4. Changes in validity graph when adding/removing components.

5 Implementation

Next, we discuss the implementation of the dynamic JavaBIP extension, during which the implementation of the JavaBIP engine has significantly changed. Let us consider first the interface of the JavaBIP engine, i.e., `BIPEngine`. In the static implementation, `BIPEngine` consisted of the following functions: (1) `register` used by a developer to register a component to the engine; (2) `inform` used by a component to inform the engine of its current state and enabled transitions; (3) `specifyGlue` used by a developer to send the glue specification to engine; (4) `start` used by a developer to start the engine thread and (5) `stop` used by a developer to stop the engine.

We updated `BIPEngine` as follows. Function `start` was removed, since the engine thread is now started automatically based on whether enough components are registered to form a valid system. We added two functions: (1) `deregister` used by a developer or the component itself (e.g., in the case of a failure) to deregister from the engine and (2) `pause` used by a developer or the component (e.g., in the case that the component is going to communicate asynchronously with other components for an amount of time) to pause synchronizations with other components. Function `register` was considerably updated, as well as function `stop` which can also be called internally by the engine in the case of an invalid system. The remaining functions were not modified. Figure 5 shows the software architecture of the JavaBIP engine. The arrows labeled `register`, `deregister`, `stop`, `specifyGlue`, and `pause` represent calls to the `BIPEngine` functions.

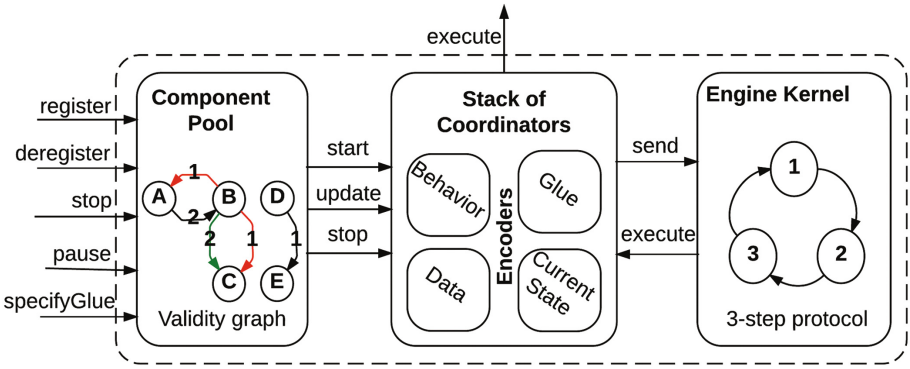

Fig. 5. Dynamic JavaBIP Engine software architecture.

The `ComponentPool` object was added, which is used as an interface to the validity graph described in Definition 4. The `ComponentPool` starts the core engine (comprising a stack of coordinators and the engine kernel), when the system becomes valid, and stops it, when the system becomes invalid. System validity is checked whenever a component is registered, deregistered or paused, excluding the cases described in Corollaries 1 and 2. Whenever a component is registered or deregistered without affecting the validity of the system, the `Component Pool` sends an update registration/deregistration event to the core engine.

The engine composes and solves the various constraints of the system. Its implementation is based on Binary Decision Diagrams (BDDs) [2], which are efficient data structures to store and manipulate Boolean formulas.[4] The imposed constraints encode information about the behavior, glue, data, and current state of the components. Current state constraints allow us to compute the enabled transitions of the component. For each type of constraints, we discuss which parts must be recomputed when registering components at runtime. There is no need to recompute these constraints when a component is paused or deregistered. Whenever constraints are recomputed, the Coordinators send these to the kernel.

The formulas that define the behavior, glue, data, and current state constraints were presented in [10]. Figure 6 summarizes the constraint computation. The white color indicates that the constraint is computed only once at system initialization. The light gray indicates that the constraint is recomputed when a component is registered. The dark gray color indicates that the constraint is recomputed during each execution cycle.

The *behavior constraint* of a component includes its ports and states. For each port, a Boolean port variable is created. Similarly, for each state, a Boolean state variable is created. Behavior constraints are built using these port and state variables. The *total behavior constraint* is computed as the conjunction of all component behavior constraints. When a component is registered, its behavior

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

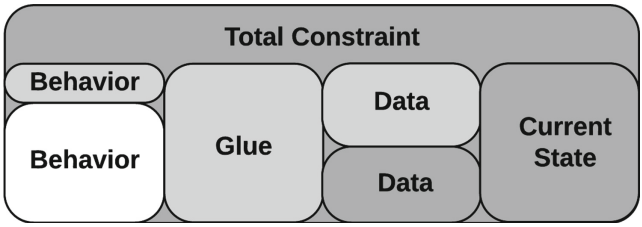

Fig. 6. Constraint computation phases.

constraint is computed and conjuncted to the total behavior constraint. When a component is deregistered, its port variables are set to *false*.

The *glue constraint* is computed by interpreting the Require and Accept macros of the glue specification. The same Boolean port variables that were previously created for the behavior constraints are used for the glue constraint as well. The glue constraint must be recomputed, in a valid system, every time a new component is registered.

For the *data constraint*, additional data variables have to be created. The data constraints represent how data is exchanged among components, i.e., which components are providing data and which components are consuming data. For each pair of components exchanging data, a data variable is created. When a component is registered, the data constraints that involve the newly arrived components are recomputed. Components exchange data at the beginning of each execution cycle of the system. Based on the exchanged data, components may disable some of the possible interactions. As a result, a subset of data constraints is recomputed at each execution cycle.

The *current state constraint* of a component is computed when a component informs of its disabled transitions due to guard evaluation. The *total current state BDD* is the conjunction of the current state constraints of all registered components. During engine execution, i.e., in a valid system, the total current state constraint is computed at each execution cycle of the engine and is further conjuncted with the total behavior constraint, the glue constraint, and the total data constraint.

The execution of a JavaBIP valid system is driven by the engine kernel applying the following protocol in a cyclic manner:

1. Upon reaching a state, all component constraints are sent to the kernel;
2. The kernel computes the *total constraint*, which is the conjunction of the total behavior, glue, current state and data constraints. Thus, it computes the possible interactions satisfying the total system constraint and picks one of them;
3. The kernel notifies the Coordinators of its decision by calling `execute`, which then notify the components to execute the necessary transitions.

Notice that a component can be registered during any step of the engine protocol. The engine, however will only include the newly registered component

in the BDD computation at the beginning of the next cycle. System validity is checked, when a component is paused or deregistered. If the system remains valid and the engine is executing the second or third step of the engine protocol, the engine sets the port variables of this component to *false* and recomputes the possible interactions.

5.1 Performance Results

We show performance results for the modular phone case study. The experiments were performed on a 3.1 GHz Intel Core i7 with 8GB RAM. We started with 5 registered components and registered up to 45 additional components. The JavaBIP models are available online[5]. Table 1 summarizes the engine's computation times and the BDD Manager peak memory usage for various numbers of components. We present and discuss three different engine times: (1) the time needed to perform a complete engine execution cycle (three-step protocol run by the Engine kernel); (2) the time needed to (partially) recompute the behavior, glue, and data BDD constraints due to the registration of a new component; (3) the time needed to add or remove a component from the component tool and check the validity of the system.

The first column shows the number of components in the system, after the registration or deregistration of a component. For instance, 10 means that a new component was registered and the total number of components in the system is now 10. The number of components is also decreased in two cases, when it is equal to 11 and equal to 29. This means that a component was deregistered or paused and the total number of components in the system is 11 or 29, respectively.

The second column shows the average engine execution time of the first 1000 engine cycles after a component registration or deregistration. The system becomes valid and the engine is started upon the registration of the 12th component. As a result, the engine execution times are equal to 0 for the first two rows of the table. If the engine had been started, for instance, after the registration of the 5th component (without the system being valid), the engine would have needed < 1 ms per execution cycle. This means that an overhead of seconds or minutes could have been added in the system's execution if more than a certain number of engine execution cycles (e.g., 100000) had been performed by the time the system became valid.

The third column of Table 1 shows the amount of time needed to recompute the behavior, glue, and data constraints of the system due to a component registration. The first two rows are equal to 0 since the system is invalid and thus, no BDD computation is required. If the engine had been started before the system became valid, the BDDs would have been recomputed upon the registration of each new component. For instance, after the registration of the 5th component, the engine would have needed 13 ms and after the registration of the 11th component, the engine would have needed additional 49 ms to recompute the BDDs.

[5] https://github.com/sbliudze/javabip-itest.

The fifth column shows the peak memory usage of the BDD manager after a component registration or deregistration.

Finally, the fourth column of Table 1 presents the amount of time needed to add or remove a component from the component pool and check for system validity. The time needed is very low, in some cases even less than 1 ms. These were the cases that system validity was not checked due to the results of Corollaries 1 and 2. The system became valid when the 12th component was registered. This required the maximum amount of time (3.654 ms), since the full graph was checked for validity, and then the core engine thread was started. Next, a component was deregistered, the system became invalid again, and the engine thread was stopped. The amount of time needed by the component pool was 2.908 ms.

Table 1. Engine times and BDD Manager peak memory usage. Times are in milliseconds and memory usage is in Megabytes.

Number of components	Time: Engine execution cycle	Time: BDD (re)computation	Time: Component pool	Memory
5	0	0	2.078	0
10	0	0	2.186	0
12	< 1	63	3.654	0.059
11	0	0	2.908	0.057
20	< 1	151	< 1	0.083
25	1.149	194	< 1	0.099
30	1.247	239	< 1	0.129
29	1.241	0	2.451	0.121
40	1.399	283	< 1	0.199
50	1.896	337	< 1	0.254

6 Related Work

Dynamicity in BIP has been studied by several authors [12,15,20]. In [12], the authors present the Dy-BIP framework that allows dynamic reconfiguration of connectors among the ports of the system. They use *history variables* to allow sequences of interactions with the same instance of a given component type. JavaBIP can emulate history variables using data. In contrast, our focus is on dynamicity due to the creation and deletion of components that is often encountered in modern software systems that are not restricted to the embedded systems domain. Additionally, the interface-based design and the modular software architecture of JavaBIP allow us to easily extend the JavaBIP implementation.

Our approach is closest to [15,21]. In [15], two extensions of the BIP model are defined: reconfigurable—similar to Dy-BIP—and dynamic, allowing

dynamic replication of components. They focus on the operational semantics of the two extensions and their properties, by studying their encodability in BIP and Place/Transition Petri nets (P/T Nets). Composition is defined through interaction models, without considering structured connectors. In contrast, our work focuses mostly on the connectivity among components, defined by Require/Accept relations. In [21], the BIP coordination mechanisms are implemented by a set of connector combinators in Haskell and Scala. Functional BIP provides combinators for managing connections in a dynamically evolving set of components. However, as in [15], such evolution must be managed by explicit actions of existing components. In contrast, the JavaBIP approach allows components to be created independently, only requiring that they be subsequently registered with the JavaBIP engine.

The Reo coordination language [33]—which realizes component coordination through circuit-like connectors built of channels and nodes—provides dedicated primitives for reconfiguring connectors by creating new channels (*Ch*), and manipulating channel ends and nodes (`split`, `join`, `hide` and `forget`). A number of papers study reconfiguration of Reo connectors. In particular, [18] provides a framework for model checking reconfigurable circuits, whereas [26] and [27] take the approach based on graph transformation techniques. The main difference between connector reconfiguration in Reo and dynamicity in JavaBIP is that, in Reo, reconfiguration operations are performed on constituent elements of the connector. Thus, in principle, such operations can affect *ongoing* interactions. This is not possible in JavaBIP, since interactions are completely atomic.

In [16,34], the authors study adaptation of open component-based systems. The underlying component and composition model inherits from the work of Arnold on synchronization vectors [4] and thereby is very close to that of BIP and other frameworks, e.g., [23]. The authors focus on fixing several types of mismatch situations, among which *name mismatch*, which occurs when the names of the sending and receiving events do not coincide (CCS and π-calculus are used to describe component behavior), and *independent evolution*, which occurs when an event on a particular interface does not have an equivalent in its counterparts interface. Contrary to BIP, synchronization and data transfer concerns are not fully separate in [16,34], since they rely on *send-receive* primitives. Thus, they use multiparty synchronization exclusively to allow broadcasting. The goal of both static and dynamic adaptation is to resolve the mismatches, based on a user-specified mapping, while avoiding deadlocks. Our work in this paper differs insofar as the BIP synchronization mechanism is blocked *anyway* (communication through direct message passing can still proceed) if the necessary components have not been registered. Thus, our main concern is not avoiding deadlock, but reducing the coordination overhead induced by the engine.

Three main types of formalisms have been studied in the literature for the specification of dynamic architectures and architecture styles [13]: (1) graph grammars, (2) process algebras, and (3) logics. Graph grammars have been used to specify reconfiguration in a dynamic architecture through the use of graph rewriting rules. Representative approaches include the Le Métayer approach [28],

where nodes plus CSP-like behavior specifications are components and edges are connectors. A different way of representing software architectures with graph grammars can be found in [24], where hyperedges with CCS labels are components and nodes are communication ports. Other graph-based approaches are summarized in [14]. None of these approaches offers tool support.

Additionally, process algebras have been used to define dynamic architectures as part of several architecture description languages (ADLs). For instance, π-calculus [32] was used in Darwin [29] and LEDA [17], CCS was used in PiLar [19], and CSP was used in Dynamic Wright [3]. In comparison with our approach, Darwin and PiLar support only binary bindings (connectors), while in Dynamic Wright and LEDA there is no clear distinction between behavior and coordination since connectors can have behavior.

Logic has also been used for the specification of dynamic software architectures and architecture styles. Alloy's first-order logic [25] was used in [22] for the specification of dynamic architectures, while the Alloy Analyzer tool was used to analyze these specifications. JavaBIP specifications can also be analyzed [7,8], however, the main focus of JavaBIP is runtime coordination, which is not offered in [22]. Configuration logics [30] were proposed for the specification of architecture styles, which however, in their current form do not capture dynamic change.

7 Conclusion and Future Work

We presented an extension of the JavaBIP framework for coordination of software components that can register, deregister and pause at runtime. To handle this type of dynamicity, JavaBIP uses a macro-notation based on first-order interaction logic that allows specifying synchronization constraints on component types. This way, a developer is not required to know the exact number of components that need to be coordinated when specifying the synchronization constraints of a system. We introduced a notion of system validity that is used to start and stop the JavaBIP engine automatically at runtime depending on whether there are enough registered components in the system so that there is at least one possible interaction. In the previous, static JavaBIP implementation, developers had to manually start and stop the engine. Starting and stopping the engine in an automatic way helps optimize JavaBIP performance since it eliminates the engine's overhead in the case of an invalid system.

JavaBIP implements the principles of the BIP component framework rooted in rigorous operational semantics. Notice, however, that currently none of the other BIP engine implementations can handle dynamic insertion and deletion of components at runtime. The functionality of pausing a component at runtime increases the incrementality of the JavaBIP engine. In our previous, static implementation, the engine had to wait for all registered components to inform in each cycle before making any computations. As a result, a single component could introduce a long delay in the system execution. In the current implementation, when a component is paused, the engine does not wait for it to inform,

but rather computes the set of enabled interaction in the system that involve only the non-paused components. JavaBIP is an open-source tool[6].

Future work includes increasing the incrementality of the engine in the following way: the engine does not have to wait for all non-paused components to inform but rather checks whether there is an enabled interaction among the components that have already informed and orders its execution. To check the enableness of interactions we plan to reuse the notion of validity graphs introduced in this paper and extend it with additional information on component ports. Additionally, we plan on extending the engine functionality to handle registration of new component types and synchronization patterns.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **C-27**(6), 509–516 (1978)
3. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998). doi:10.1007/BFb0053581
4. Arnold, A.: Synchronized behaviours of processes and rational relations. Acta Informatica **17**, 21–29 (1982)
5. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)
6. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), pp. 3–12, September 2006, Invited talk
7. Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J.: D-Finder: a tool for compositional deadlock detection and verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02658-4_45
8. Bliudze, S., Cimatti, A., Jaber, M., Mover, S., Roveri, M., Saab, W., Wang, Q.: Formal verification of infinite-state BIP models. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 326–343. Springer, Cham (2015). doi:10.1007/978-3-319-24953-7_25
9. Bliudze, S., Mavridou, A., Szymanek, R., Zolotukhina, A.: Coordination of software components with BIP: application to OSGi. In: Proceedings of the 6th International Workshop on Modeling in Software Engineering, MiSE 2014, pp. 25–30. ACM, New York (2014)
10. Bliudze, S., Mavridou, A., Szymanek, R., Zolotukhina, A.: Exogenous coordination of concurrent software components with JavaBIP. Software: Practice and Experience (2017). Early view: http://dx.doi.org/10.1002/spe.2495
11. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: From high-level component-based models to distributed implementations. In: Proceedings of the Tenth ACM International Conference on Embedded Software, EMSOFT 2010, pp. 209–218. ACM, New York (2010)

[6] github.com/sbliudze/javabip-core, github.com/sbliudze/javabip-engine.

12. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling dynamic architectures using Dy-BIP. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (eds.) SC 2012. LNCS, vol. 7306, pp. 1–16. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30564-1_1
13. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, WOSS 2004, pp. 28–33. ACM, New York (2004)
14. Bruni, R., Bucchiarone, A., Gnesi, S., Melgratti, H.: Modelling dynamic software architectures using typed graph grammars. Electron. Notes Theoret. Comput. Sci. **213**(1), 39–53 (2008)
15. Bruni, R., Melgratti, H., Montanari, U.: Behaviour, interaction and dynamics. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 382–401. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54624-2_19
16. Cámara, J., Salaün, G., Canal, C.: Composition and run-time adaptation of mismatching behavioural interfaces. J. Univ. Comput. Sci. **14**(13), 2182–2211 (2008)
17. Canal, C., Pimentel, E., Troya, J.M.: Specification and refinement of dynamic software architectures. In: Donohoe, P. (ed.) Software Architecture. ITIFIP, vol. 12, pp. 107–125. Springer, Boston (1999). doi:10.1007/978-0-387-35563-4_7
18. Clarke, D.: A basic logic for reasoning about connector reconfiguration. Fundamenta Informaticae **82**(4), 361–390 (2008)
19. Cuesta, C.E., de la Fuente, P., Barrio-Solárzano, M.: Dynamic coordination architecture through the use of reflection. In: Proceedings of the 2001 ACM Symposium on Applied Computing, pp. 134–140. ACM (2001)
20. Di Giusto, C., Stefani, J.-B.: Revisiting glue expressiveness in component-based systems. In: De Meuter, W., Roman, G.-C. (eds.) COORDINATION 2011. LNCS, vol. 6721, pp. 16–30. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21464-6_2
21. Edelmann, R., Bliudze, S., Sifakis, J., Functional, B.I.P.: Embedding connectors in functional programming languages. J. Logical Algebraic Methods Program. (2017) (Under review)
22. Georgiadis, I., Magee, J., Kramer, J.: Self-organising software architectures for distributed systems. In: Proceedings of the First Workshop on Self-Healing Systems, pp. 33–38. ACM (2002)
23. Henrio, L., Madelaine, E., Zhang, M.: A theory for the composition of concurrent processes. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 175–194. Springer, Cham (2016). doi:10.1007/978-3-319-39570-8_12
24. Hirsch, D., Inverardi, P., Montanari, U.: Graph grammars and constraint solving for software architecture styles. In: Proceedings of the Third International Workshop on Software Architecture, pp. 69–72. ACM (1998)
25. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. (TOSEM) **11**(2), 256–290 (2002)
26. Koehler, C., Costa, D., Proença, J., Arbab, F.: Reconfiguration of Reo connectors triggered by dataflow. ECEASST **10** (2008)
27. Krause, C., Maraikar, Z., Lazovik, A., Arbab, F.: Modeling dynamic reconfigurations in Reo using high-level replacement systems. Sci. Comput. Program. **76**(1), 23–36 (2011)
28. Le Métayer, D.: Describing software architecture styles using graph grammars. IEEE Trans. Softw. Eng. **24**(7), 521–533 (1998)
29. Magee, J., Kramer, J.: Dynamic structure in software architectures. ACM SIGSOFT Softw. Eng. Notes **21**(6), 3–14 (1996)

30. Mavridou, A., Baranov, E., Bliudze, S., Sifakis, J.: Configuration logics: modeling architecture styles. J. Logical Algebraic Methods Program. **86**(1), 2–29 (2017)
31. Mavridou, A., Rutz, V., Bliudze, S.: Coordination of dynamic software components with JavaBIP. Technical report (2017), https://arxiv.org/abs/1707.09716
32. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Inf. Comput. **100**(1), 1–40 (1992)
33. Papadopoulos, G.A., Arbab, F.: Configuration and dynamic reconfiguration of components using the coordination paradigm. Future Gener. Comput. Syst. **17**(8), 1023–1038 (2001)
34. Poizat, P., Salaün, G.: Adaptation of open component-based systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 141–156. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72952-5_9