Coordination of Dynamic Software Components with JavaBIP

Anastasia Mavridou, Valentin Rutz, Simon Bliudze

May 12, 2021

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 than 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 coloring edges 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.

```
@TechReport{DynamicJavaBIP,
Author = {Anastasia Mavridou and Valentin Rutz and Simon Bliudze},
Title = {Coordination of {D}ynamic {S}oftware {C}omponents with {J}ava{BIP}},
note     = {available at \url{https://arxiv.org/abs/1707.09716}},
Year = {2017},
Eprint = {arXiv:1707.09716}
}
```

Contents

1 Introduction

We have previously introduced JavaBIP [8, 9] 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 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 [6, 7] ensuring the correctness of JavaBIP systems.

The previous implementation of JavaBIP [9] 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 [4, 5, 12] allows to add or remove components on-the-fly, including DyBIP presented in [13] 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 to start and stop the engine 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 report is structured as follows. Section 2 presents the JavaBIP framework. Section 3 describes our motivating case study. Section 5 presents the notion of JavaBIP system validity and the construction of validity graphs. Section 4 presents the interaction logic and the macro-notation used to specify JavaBIP synchronization constraints on component types. Section 6 describes the implemented software architecture and presents performance results. Section 7 discusses related work. Section 8 summarizes the results and future work directions.

2 The JavaBIP Framework

JavaBIP implements the BIP (Behavior-Interaction-Priority) coordination mechanism [4], 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 [6], ESST or nuXmv [7]. Other analyses can be performed using any tool for which a model transformation from BIP is available.

3 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].

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

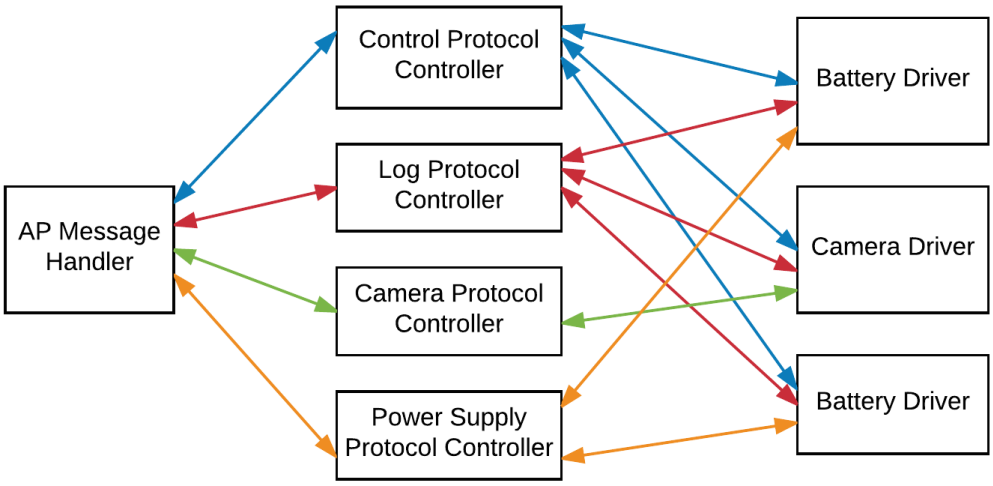

Figure 1: Modular phone architecture.

```
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)
```

Figure 2: Hierarchical decomposition of the `Application Layer` into components.

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

3.1 Componentization and Interaction Model

We have used *architecture diagrams* [30] to model the architecture style of the case study, which is shown in Fig. 3. An architecture style defines the set of all possible architectures for any number of components in the system. An architecture diagram consists of a set of *component types*, with associated cardinality constraints representing the expected number of instances of each component type and a set of *connector motifs*. The boxes in Fig. 3 represent the atomic component types of the case style, which are the following: `AP Request Worker`, `AP Response Worker`, `AP Message Worker`, `Control Protocol Controller`, `AP Receiver Fifo`, `Log Protocol Controller`, `Camer Protocol Controller`, `Control Connect Handler`, `Power Supply Protocol Controller`, `Control Disconnect Handler`, `Log Handler`, `Camera Capture Handler`, `Camera Stream Handler`, `Power Supply Handler`. The cardinalities of these component types are shown in Fig. 3 in the upper left corner of the corresponding boxes. For instance, the cardinality of the `AP Request Worker` component type is 1, while the cardinality of the `Camera Capture Handler` component type is n. The valuation of the n parameter depends on the number of cameras attached on the phone.

Figure 3 contains a set of connector motifs, which define the glue of the case study, i.e., the interaction model. Each connector motif defines a set of BIP connectors [4], which are non-empty sets of *port types*. Each connector motif end has two associated constraints: *multiplicity* and *degree*, represented as a pair $m : d$. Multiplicity constrains the number of instances of the port type that must participate in a connector defined by the motif; degree constrains the number of connectors attached to any instance of the port type. Cardinalities, multiplicities and degrees are either natural numbers or intervals. In this case study we have used only natural numbers.

Let us consider, for instance, the connector motif that connects the port type `receive` of `AP Message Worker` with the port type `rm` of `AP Receiver Fifo`. The associated constraint pair of each connector motif end is equal to $1 : 1$. This means a conforming architecture of the style will include a binary BIP connector attached to the port instance `receive` of the component instance of `AP Message Worker` and to the port instance `rm` of the component instance of `AP Receiver Fifo`. This binary BIP connector represents a synchronization of the actions `rm` and `receive` of the corresponding component instances. Lets us now consider the connector motif that connects the port type `send_log` of `Log Handler` with the port type `rcvFromDriver` of `Log Protocol Controller`. The associated constraint pair of the connector motif end attached to `send_log` is equal to $1 : 1$, while the constraint pair of the connector motif end attached to `rcvFromDriver` is equal to $1 : n$, where n is the cardinality of `Log Handler`. A conforming architecture that contains n instances of `Log Handler` will also contain n binary connectors, each connecting a distinct component instance of `Log Handler` to the unique component instance of `Log Protocol Controller`.

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

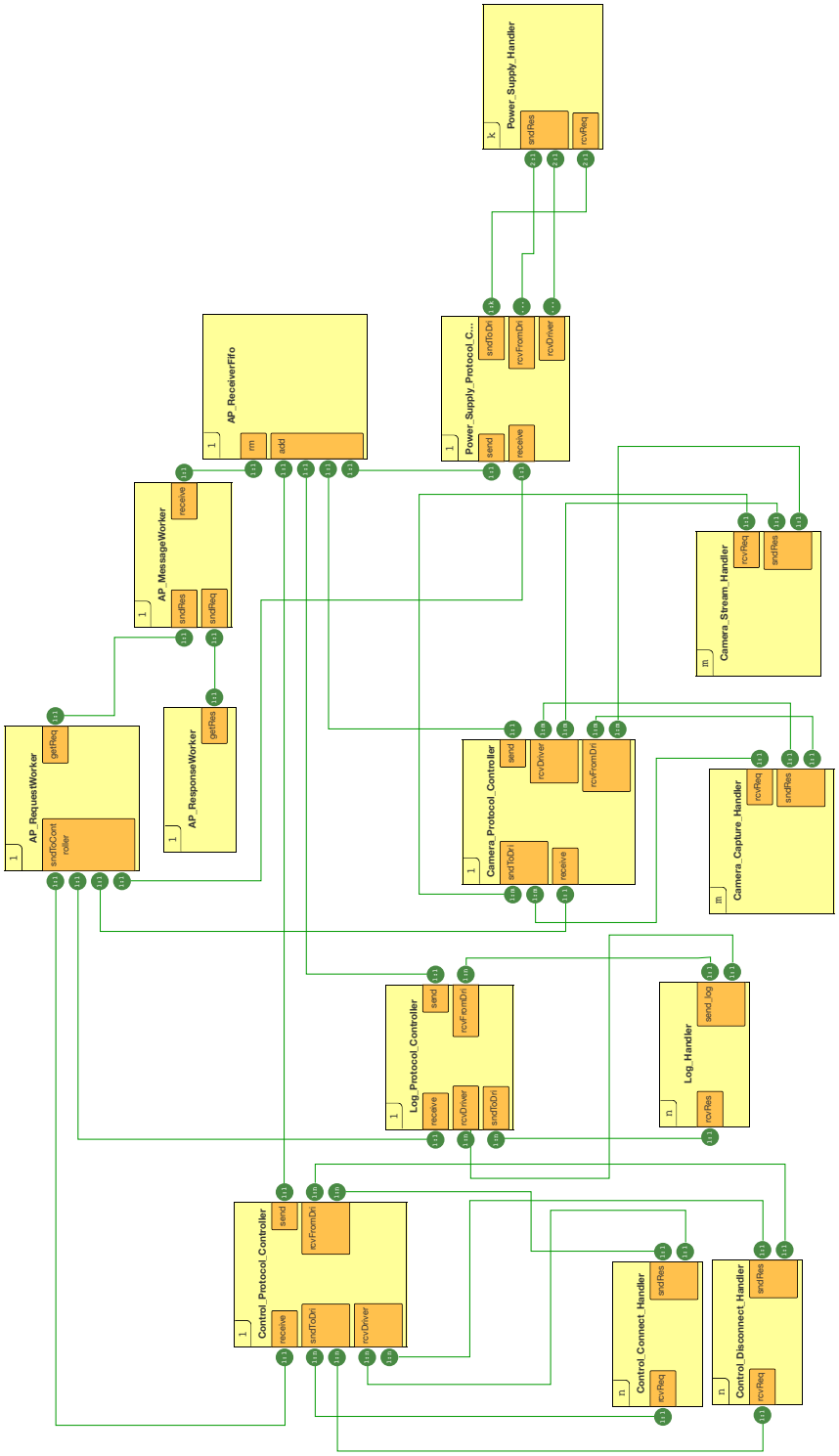

Figure 3: Modular phone architecture style.

4 Interaction Logic and Macro-notation

The glue specification is defined in JavaBIP through a macro-notation, similar to the one introduced in [13], 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. In the next subsections, we present the propositional and first-order interaction logic, as well as the JavaBIP Require/Accept macro-notation, which is based on first-order interaction logic.

4.1 Propositional Interaction Logic

The propositional interaction logic (PIL), studied in [10, 11], is a Boolean logic used to characterize a *configuration*, i.e., a non-empty set of interactions among components on a global set of ports P. We assume that P is given and $a \subseteq P$.

Definition 1. *An interaction a is a set of ports $a \subseteq P$ such that $a \neq \emptyset$.*

Syntax 1. *The propositional interaction logic is defined by the grammar:*

$$\phi ::= \mathit{true} \mid p \mid \overline{\phi} \mid \phi \vee \phi\,, \qquad \mathit{with\ any}\ p \in P. \tag{1}$$

Conjunction is defined as follows: $\phi_1 \wedge \phi_2 \stackrel{def}{=} \overline{(\overline{\phi_1} \vee \overline{\phi_2})}$. Implication is defined as follows: $\phi_1 \Rightarrow \phi_2 \stackrel{def}{=} \overline{\phi_1} \vee \phi_2$. To simplify the notation, we omit conjunction in monomials, e.g., writing sr_1r_2 instead of $s \wedge r_1 \wedge r_2$.

Semantics 1. *The meaning of a PIL formula ϕ is defined by the following satisfaction relation. Let γ be a non-empty configuration. We define: $\gamma \models \phi$ iff for all $a \in \gamma$, ϕ evaluates to true for the valuation induced by a: $p = true$, for all $p \in a$ and $p = false$, for all $p \notin a$.*

The operators meet the usual Boolean axioms and the additional axiom $\bigvee_{p \in P} p = \mathit{true}$ meaning that interactions are non-empty sets of ports.

Example 1. *Consider a* Star architecture*, where a single component C acts as the center, and three other components S_1, S_2, S_3 communicate with the center through binary synchronizations. Component C has a port p and all other components have a single port q_i $(i = 1, 2, 3)$. The corresponding PIL formula is: $pq_1\overline{q_2}\,\overline{q_3} \vee p\overline{q_1}\,q_2\overline{q_3} \vee p\overline{q_1}\,\overline{q_2}\,q_3$.*

4.2 First-order Interaction Logic

We extend the propositional interaction logic presented in Subsection 4.1 with quantification over components to define interactions independently from the number of component instances. This extension is particularly useful because, in practice, systems are built from multiple component instances of the same component type. A first-order interaction logic was also presented in [13] with additional history variables. We make the following assumptions:

- A finite set of component types $\mathcal{T} = \{T_1, \ldots, T_n\}$ is given. Instances of a component type have the same interface and behavior. We write $c{:}T$ to denote that a component c is of type T.

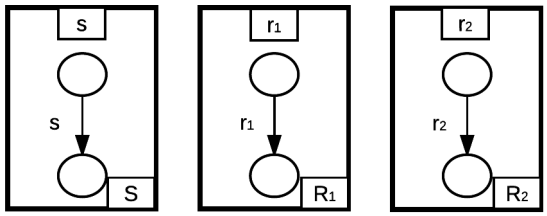

Figure 4: Three BIP components

- The interface of each component type has a distinct set of ports. We denote by $T.p$ the *port type* p, i.e., a port belonging to the interface of type T. We write $T.P$ to denote the set of port types of component type T and $\mathcal{T}.P$ to denote the set of port types of all component types. We write $c.p$, for a component $c{:}T$, to denote the *port instance* of type $T.p$.

Let ϕ denote any formula in PIL.

Syntax 2. *The first-order interaction logic (FOIL) is defined by the grammar:*

$$\Phi ::= \mathit{true} \mid \phi \mid \overline{\Phi} \mid \Phi \vee \Phi \mid \exists c : T\big(Pr(c)\big).\Phi\,, \tag{2}$$

where T is a component type, which represents a set of component instances with identical interfaces and behaviour. Variable c ranges over component instances and must occur in the scope of a quantifier. $Pr(c)$ is some set-theoretic predicate on c (omitted when $Pr = \mathit{true}$).

Additionally, we define the usual notation for the universal quantifier:

$$\forall c{:}T\big(\mathrm{Pr}(c)\big).\Phi \stackrel{def}{=} \nexists c{:}T\big(\mathrm{Pr}(c)\big).\overline{\Phi}\,.$$

Semantics 2. *The semantics is defined for closed formulas, where, for each variable in the formula, there is a quantifier over this variable in a higher nesting level. We assume that the finite set of component types $\mathcal{T} = \{T_1, \ldots, T_n\}$ is given. Models are pairs $\langle \mathcal{B}, \gamma \rangle$, where $\mathcal{B}$ is a set of component instances of types from $\mathcal{T}$ and γ is a configuration on the set of ports P of these components. For quantifier-free formulas, the semantics is the same as for PIL formulas. For formulas with quantifiers, the satisfaction relation is defined as follows:*

$$\langle \mathcal{B}, \gamma \rangle \models \exists c : T\big(Pr(c)\big).\Phi\,, \qquad \textit{iff } \gamma \models \bigvee_{c':T \in B \wedge Pr(c')} \Phi[c'/c],$$

where $c' : T$ ranges over all component instances of type $T \in \mathcal{T}$ and $\Phi[c'/c]$ is obtained by replacing all occurrences of c in Φ by c'.

Example 2. *Consider three components shown in Fig. 4: a sender S and two receivers R_1, R_2. The sender has a port s for sending messages and each receiver has a port r_i $(i = 1, 2)$ for receiving them. We consider the following four coordination schemes:*

- Rendezvous *ensures strong synchronization between S and all R_i. Rendezvous is specified by a single interaction involving all ports represented by the monomial sr_1r_2. This interaction can occur only if all of the components are in states enabling transitions labelled, respectively, by s, r_1 and r_2.*

- Broadcast *allows all interactions involving S and any (possible empty) subset of R_i. Broadcast is represented by the formula s, which can be expanded to $s\overline{r_1}\,\overline{r_2} \vee sr_1\overline{r_2} \vee s\overline{r_1}r_2 \vee sr_1r_2$. These interactions can only occur if S is in a state enabling s. Each R_i participates in an interaction only if it is in a state enabling r_i.*
- Atomic broadcast *ensures that either all or none of the receivers are involved in the interaction. Atomic broadcast is be characterised by the formula $s\overline{r_1}\,\overline{r_2} \vee sr_1r_2$. The sr_1r_2 interaction corresponds to a strong synchronization among the sender and all receivers.*

Example 3. *The Star architecture (Ex. 1) can be expressed in FOIL for any number of components of type S as follows:*

$$\exists c{:}C.\ \forall s{:}S.\ (c.p\ s.q) \wedge\ \forall s'{:}S\ (s \neq s').\ (\overline{s.q\ s'.q}\ ) \wedge \forall c' : C(c = c').\ true.$$

4.3 JavaBIP Require/Accept Macro-notation Based on FOIL

JavaBIP relies on component types, rather than on component instances for the definition of synchronization constraints. All instances of a given component type are restricted with the same set of synchronization constraints.

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. Similarly to [13], 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 examples.

4.3.1 The Require Macro

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

$$T_1.p\ \ \textbf{Require}\ \ T_2.q\ \ \equiv\ \ \forall c_1{:}T_1.\ \exists c_2{:}T_2.\ \forall c_3{:}T_2\ (c_2 \neq c_3).\ \big(c_1.p\ \Rightarrow\ c_2.q\ \overline{c_3.q}\big)\,,$$

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. In comparison with [13], we have opted for a macro-notation where the cardinality is explicit: should two instances of the same port type be required, this is specified by explicitly putting the required port type twice:

$$\begin{aligned} T_1.p\ \ \textbf{Require}\ \ T_2.q\ T_2.q\ \ \equiv\ \ \forall c_1{:}T_1.\ \exists c_2, c_3{:}T_2.\ \forall c_4{:}T_2\ (c_2 \neq c_3 \neq c_4).\\ \big(c_1.p\ \Rightarrow\ c_2.q\ c_3.q\ \overline{c_4.q}\big)\,, \end{aligned}$$

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. We define:

$$\begin{aligned} T_1.p\ \ \textbf{Require}\ \ T_2.q\ T_2.q\ ;\ T_2.r\ \ \equiv\ \ \forall c_1{:}T_1.\\ \Big(\exists c_2, c_3{:}T_2.\ \forall c_4{:}T_2\ (c_2 \neq c_3 \neq c_4).\ \big(c_1.p\ \Rightarrow\ c_2.q\ c_3.q\ \overline{c_4.q}\big)\\ \bigvee\ \exists c_2{:}T_2.\ \forall c_3{:}T_2(c_2 \neq c_3).\ \big(c_1.p\ \Rightarrow\ c_2.r\ \overline{c_3.r}\big)\ \Big)\,, \end{aligned}$$

which 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.

4.3.2 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 optional. Let $T^1, T^2 \in \mathcal{T}$ be two component types. The following:

$$T_1.p \;\; \textbf{Accept} \;\; T_2.q \quad \equiv \quad \forall c_1{:}T_1.\; \left(\bigwedge_{T.r \in \mathcal{T}.P \setminus \{T_2.q\}} \forall c{:}T.\; (c_1.p \Rightarrow \overline{c.r}) \right) ,$$

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

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

Example 4. *The synchronization constraints of the Star architecture (Ex. 3) are specified by the following combination of macros:*

`S.q` **Require** `C.p`		`S.q` **Accept** `C.p`	
`C.p` **Require** `S.q`		`C.p` **Accept** `S.q`	,

Example 5. *Let us now consider a more general example to illustrate the expressiveness of the synchronization constraints. Assume that there are five component types* `A, B, C, D, E` *with port types* `a, b, c, d, e`*, respectively. Through the Require macros, we enforce the following five 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 synchronisations with any possible combination of ports* `A.a, B.b, C.c`*); 4)* `D.d` *requires synchronization with a single instance of* `E.e` *and 5)* `E.e` *does not require synchronizations with other ports (however it accepts synchronisations with any number of ports* `D.d`*). Notice that by the combination of the first two require macros, a synchronisation 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 synchronisation.*

`A.a`	**Require**	`B.b B.b`		`A.a` **Accept**	`A.a B.b C.c`
`B.b`	**Require**	`A.a C.c` ; `C.c C.c`		`B.b` **Accept**	`A.a B.b C.c`
`C.c`	**Require**	–		`C.c` **Accept**	`A.a B.b C.c`
`D.d`	**Require**	`E.e`		`D.d` **Accept**	`E.e`
`E.e`	**Require**	–		`E.e` **Accept**	`D.d`

5 Defining System Validity

In the previous, static JavaBIP implementation, a developer would first register all components to the engine and then start the engine manually. Since, in the dynamic JavaBIP implementation, components may register or deregister on the fly, we need 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 atomic components, BIP systems and valid BIP systems.

Definition 2 (Component). *A component B is a Finite State Machine 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 3 (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 a Finite State Machine $(Q, \gamma, \rightarrow)$, 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 4 extends Def. 3 to describe a valid BIP system. System validity is defined from the perspective of starting or stopping the JavaBIP engine, which orchestrates component interaction (synchronization of component actions). Notice that even if a system is not valid according to Def. 4, JavaBIP components can be communicating in an asynchronous manner.

Definition 4 (Valid BIP System). *A BIP system $(Q, \gamma, \rightarrow)$ is valid iff $\gamma \neq \emptyset$.*

Remark 5.1. *In Def. 2 and Def. 3, for the sake of simplicity, we omit the presentation of data-related aspects. However, it should be noted that the full JavaBIP [9] allows data variables within components. In such cases, component transitions can be guarded by Boolean predicates on data variables. Notice that in Def. 4 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 4 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 5 (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. 5 shows the validity graph of the Star architecture defined in the glue specification of Ex. 4. The two vertices represent the two component types `S, D` of the example. Since, component type `S` requires synchronization with exactly one instance of component type `C` (cardinality = 1), there is an edge from vertex `S` to vertex `C` labeled by a counter initialised to 1. Analogously, since, component type `C` requires synchronization with exactly one instance of component type `S` (cardinality = 1), there is an edge from vertex `C` to vertex `S` labeled by a counter initialised to 1.

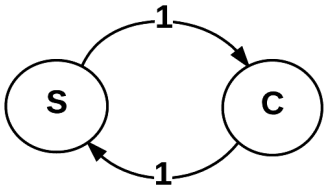

Figure 5: Validity graph of Ex. 4.

Fig. 6 shows the validity graph that corresponds to the glue specification of Ex. 5. There are five vertices `A, B, C, D, E` each representing a component type of the example. Let us consider the second and third Require macros of the example. In the second Require macro notice the two OR-causes, each represented by edges with different colors (red and green) in Fig. 6. In the first OR-cause, `B` requires one instance of `A` and one instance of `C`, represented by the red edges in Fig. 6 both labeled by 1 since both cardinalities are equal to 1. In the second OR-cause, `B` requires two instances of `C`, represented by the green edge in Fig. 6 labeled by 2. In the third Require macro of the example, `C` does not require synchronization with any other component of the system, and thus, there is no outgoing edge from vertex `C`.

Fig. 7 shows the validity graph of the modular phone case study. The are 13 vertices, each one representing an atomic component type of the case study (Fig. 2). These are the following: `AP Request Worker`, `AP Response Worker`, `AP Message Worker`, `AP Receiver Fifo`, `Control Protocol Controller`, `Log Protocol Controller`, `Camera Protocol Controller`, `Power Supply Protocol Controller`, `Control Connect Handler`, `Control Disconnect Handler`, `Log Handler`, `Camera Capture Handler`, `Camera Stream Handler`, and `Power Supply Handler`. Due to space limitations, we have substituted the full names with their acronyms. For instance, we have substituted `AP Message Handler` by `APMW`. In case of acronym conflicts, we added more letters, e.g., we have substituted `AP Request Worker` by `APReqW`, and `AP Response Worker` by `APResW`.

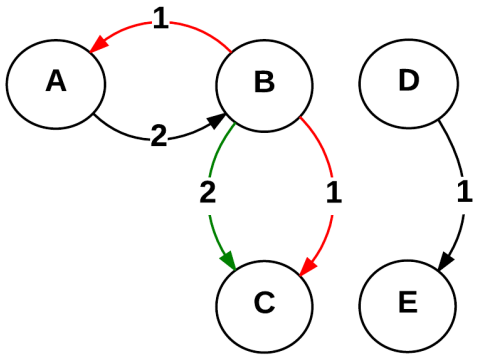

Figure 6: Validity graph of Ex. 5.

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 the Appendix).

`PSPC.rcvFromDri` **Require** `PSH.sndRes PSH.sndRes`
`APRF.add` **Require** `ConPC.snd; CamPC.snd; PSPC.snd; LPC.snd`

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

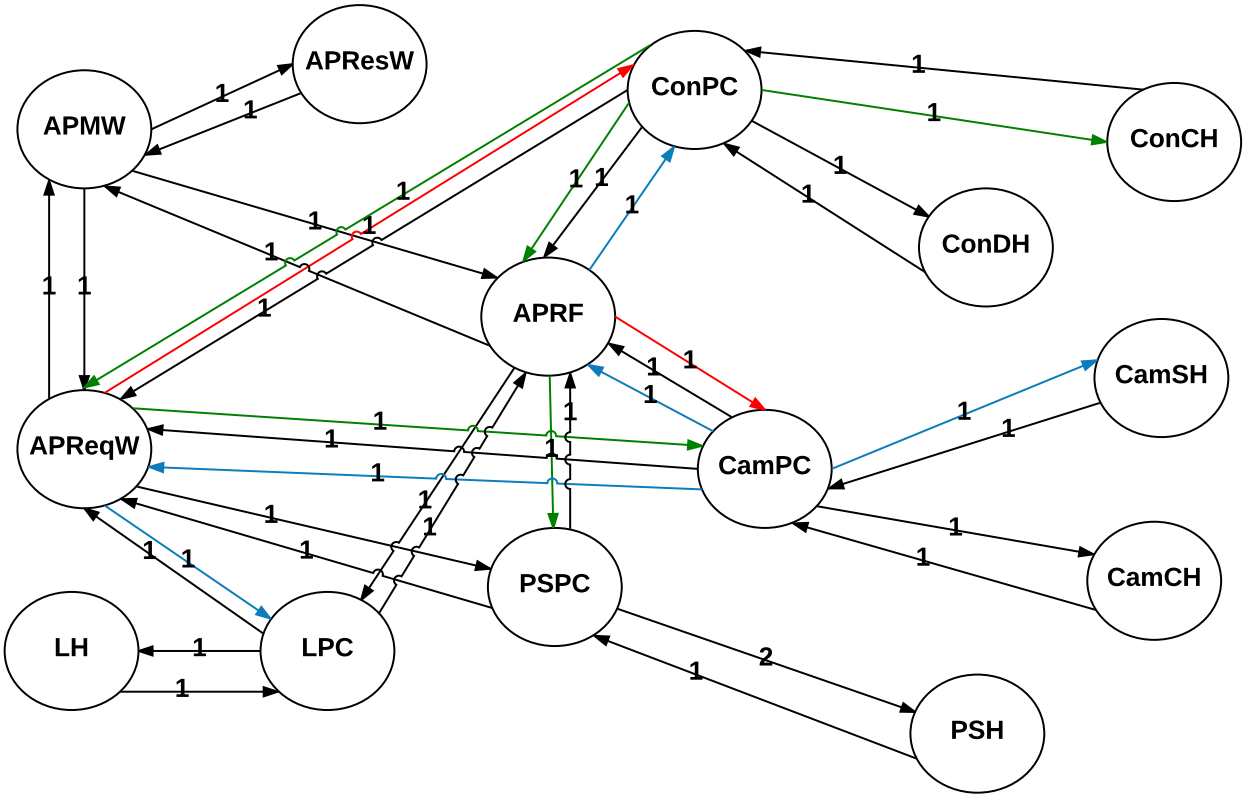

Figure 7: Validity graph of the Modular Phone case study.

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. 7, edges with different colors are also represented by different line styles.

Definition 6 (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 5.2 (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*.*

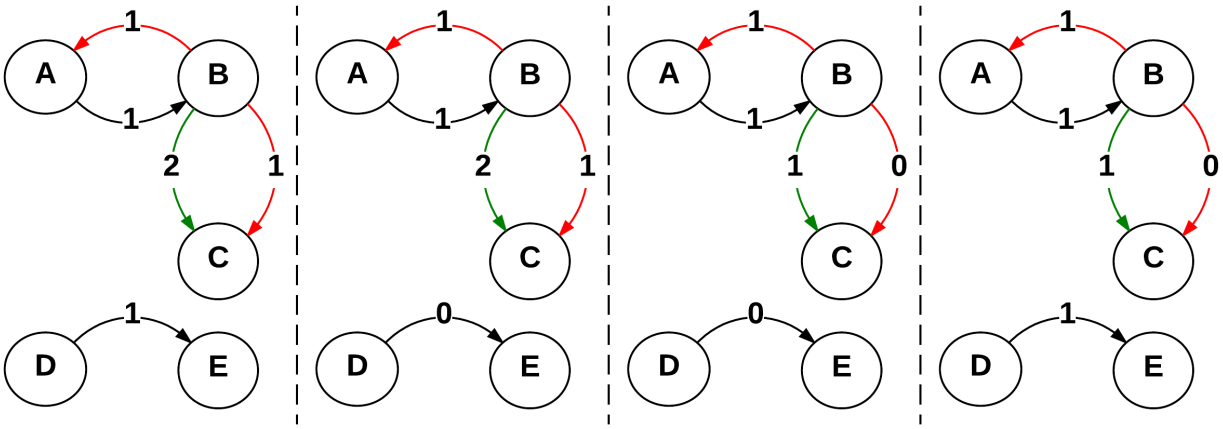

Figure 8: Changes in validity graph when adding/removing components.

Proof. Necessity: Since $\gamma \neq \emptyset$, there exists at least one interaction $a \in \gamma$. This means that there exists a `Require` macro such that the effect is the type of a port instance $p \in a$ and the causes contain an OR-cause OR_p that consists of a subset of the types of port instances in a. We denote this subset by Q. For each port type $q \in Q$, there exists at least one instance of $q \in a$. The cardinality of each port type in OR_p is equal to the number of corresponding instances in a. By Definition 5, there exists a vertex v in the validity graph that corresponds to the component type that contains p and a set of outgoing edges. Let us denote V_p the subset of outgoing edges of v that correspond to OR_p, which are colored the same. Since $a \in \gamma$, this implies that all counters of V_p are equal to or less than 0 and there is at least a registered instance of the component type that contains p.

Sufficiency: We know that there exists a vertex with a registered component instance and a color such that the counters of all the outgoing edges of that color are equal to or less than 0. Consider a `Require` macro with an OR-cause OR_c that corresponds to this color. Since all the counters of edges of this color are equal to 0, we know that there exists a sufficient number of registered instances of every type in OR_c . Since the effect of the `Require` macro is also registered, there is an enabled interaction involving component instances corresponding to the effect and the elements of OR_c . This implies that the system is valid, which concludes the proof. □

Example 6. *Let us now consider a system with the glue specification of Ex. 5. Fig. 8 present the changes of the corresponding validity graph (see Fig. 6) when 1) an instance of* `B` *is registered; 2) an instance of* `E` *is registered; 3) an instance of* `C` *is registered and 4) the instance of* `E` *is deregistered. Notice that the system becomes valid when an instance of* `E` *is registered and becomes invalid when the instance of* `E` *is deregistered.*

To start and stop the engine, we determine first whether the system is valid by using Prop. 5.2. Nevertheless, we do not need to check system validity every time a component registers/deregisters/pauses. Corollaries 5.3 and 5.4 define such cases.

Corollary 5.3. *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.*

Proof. It follows directly from Proposition 5.2 and Definition 6.

□

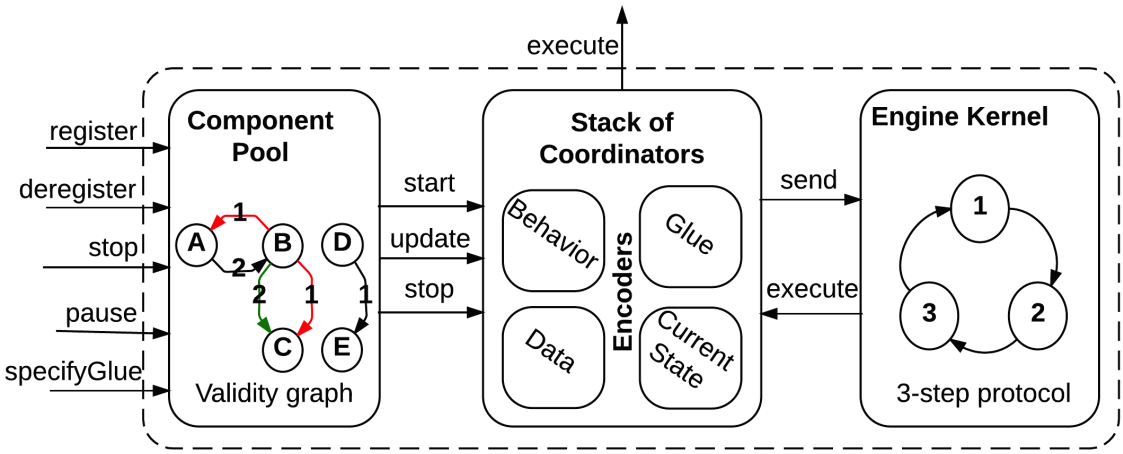

Figure 9: Dynamic JavaBIP Engine software architecture.

Corollary 5.4. *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.*

Proof. It follows directly from Proposition 5.2 and Definition 6. □

6 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 been updated. Figure 9 shows the software architecture of the JavaBIP engine. The arrows labeled `register`, `deregister`, `stop`, `specifyGlue`, and `pause` represent calls to the `BIPEngine` functions.

The `ComponentPool` object was added, which is used as an interface to the validity graph described in Def. 5. 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 Cor. 5.3 and 5.4. 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.

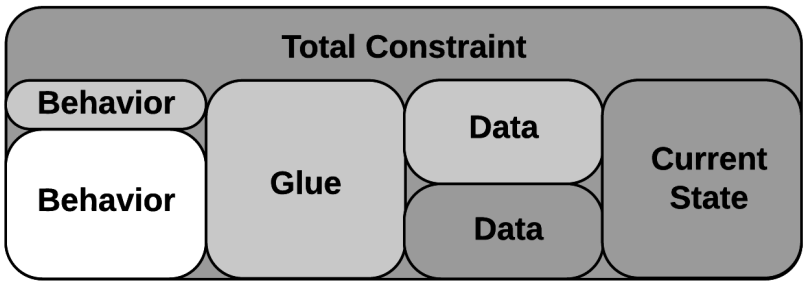

Figure 10: Constraint computation phases.

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.[3] 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 [9]. Figure 10 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 the ports and states of the component. 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 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

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

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.

6.1 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[4]. 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 12^{th} 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 5^{th} 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.

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

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

For instance, after the registration of the 5^{th} component, the engine would have needed 13 ms and after the registration of the 11^{th} component, the engine would have needed additional 49 ms to recompute the BDDs. 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 Cor. 5.3 and 5.4. The system became valid when the 12^{th} 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.

7 Related Work

Dynamicity in BIP has been studied by several authors [13, 16, 20]. In [13], 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 [16] and [21]. In [16], 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 [16], 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.

Three main types of formalisms have been studied in the literature for the specification of dynamic architectures and architecture styles [14]: 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 [23], where hyperedges with CCS labels are components and nodes are communication ports. Other graph-based approaches are summarized in [15]. 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 [6, 7], however, the main focus of JavaBIP is runtime coordination, which is not offered in [22]. Configuration logics [31] were proposed for the specification of architecture styles, which however, in their current form do not capture dynamic change.

8 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. Additionally, we introduced a notion of system validity, which we use 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 enabled synchronization among them. 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 none of the current BIP engine implementations can handle dynamic insertion and deletion of components at runtime. The functionality of pausing a component at runtime makes the implementation of the JavaBIP engine more incremental. 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[5].

In the future, we plan to work towards 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] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

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

[3] Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. *Fundamental Approaches to Software Engineering*, pages 21–37, 1998.

[4] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, 2011.

[5] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In 4^{th} *IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM06)*, pages 3–12, September 2006. Invited talk.

[6] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. DFinder: A tool for compositional deadlock detection and verification. In *Computer Aided Verification*, pages 614–619. Springer, 2009.

[7] Simon Bliudze, Alessandro Cimatti, Mohamad Jaber, Sergio Mover, Marco Roveri, Wajeb Saab, and Qiang Wang. Formal verification of infinite-state BIP models. In *International Symposium on Automated Technology for Verification and Analysis*, pages 326–343. Springer, 2015.

[8] Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. Coordination of software components with BIP: Application to OSGi. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, MiSE 2014, pages 25–30, New York, NY, USA, 2014. ACM.

[9] Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. Exogenous coordination of concurrent software components with JavaBIP. *Software: Practice and Experience*, page n/a, 2017. Early view: `http://dx.doi.org/10.1002/spe.2495`.

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

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

[11] Simon Bliudze and Joseph Sifakis. Causal semantics for the algebra of connectors. *Formal Methods in System Design*, 36(2):167–194, June 2010.

[12] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 209–218, New York, NY, USA, 2010. ACM.

[13] Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling dynamic architectures using Dy-BIP. In *Software Composition (SC 2012)*, volume 7306 of *LNCS*, pages 1–16. Springer, 2012.

[14] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, WOSS '04, pages 28–33, New York, NY, USA, 2004. ACM.

[15] Roberto Bruni, Antonio Bucchiarone, Stefania Gnesi, and Hernán Melgratti. Modelling dynamic software architectures using typed graph grammars. *Electronic Notes in Theoretical Computer Science*, 213(1):39 – 53, 2008.

[16] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Behaviour, interaction and dynamics. In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, volume 8373 of *LNCS*, pages 382–401. Springer, 2014.

[17] Calos Canal, Ernesto Pimentel, and José M Troya. Specification and refinement of dynamic software architectures. In *Software Architecture*, pages 107–125. Springer, 1999.

[18] Dave Clarke. A basic logic for reasoning about connector reconfiguration. *Fundamenta Informaticae*, 82(4):361–390, 2008.

[19] Carlos E Cuesta, Pablo de la Fuente, and Manuel Barrio-Solárzano. Dynamic coordination architecture through the use of reflection. In *Proceedings of the 2001 ACM symposium on Applied computing*, pages 134–140. ACM, 2001.

[20] Cinzia Di Giusto and Jean-Bernard Stefani. Revisiting glue expressiveness in component-based systems. In *COORDINATION 2011*, pages 16–30. Springer, 2011.

[21] Romain Edelmann, Simon Bliudze, and Joseph Sifakis. Functional BIP: Embedding connectors in functional programming languages. *Journal of Logical and Algebraic Methods in Programming*, 2017. Under review.

[22] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, pages 33–38. ACM, 2002.

[23] Dan Hirsch, Paola Inverardi, and Ugo Montanari. Graph grammars and constraint solving for software architecture styles. In *Proceedings of the third international workshop on Software architecture*, pages 69–72. ACM, 1998.

[24] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *Software Engineering, IEEE Transactions on*, 21(4):373–386, 1995.

[25] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[26] Christian Koehler, David Costa, José Proença, and Farhad Arbab. Reconfiguration of Reo connectors triggered by dataflow. *ECEASST*, 10, 2008.

[27] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23–36, 2011.

[28] Daniel Le Métayer. Describing software architecture styles using graph grammars. *Software Engineering, IEEE Transactions on*, 24(7):521–533, 1998.

[29] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *ACM SIGSOFT Software Engineering Notes*, 21(6):3–14, 1996.

[30] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. Architecture diagrams: A graphical language for architecture style specification. In *Proceedings of the 9th Interaction and Concurrency Experience (ICE)*, pages 83–97, August 2016.

[31] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. Configuration logics: Modeling architecture styles. *Journal of Logical and Algebraic Methods in Programming*, 86(1):2–29, 2017.

[32] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and computation*, 100(1):1–40, 1992.

[33] George A. Papadopoulos and Farhad Arbab. Configuration and dynamic reconfiguration of components using the coordination paradigm. *Future Generation Computer Systems*, 17(8):1023–1038, 2001.

A Complete glue specification of the modular phone case study

The glue specification must be provided in an XML file. 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. `require` constraints have a set of options for causes, i.e., the OR-causes explained in Subsection 3.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<glue>
  <accepts>
    <accept>
      <effect id="sndToDri" specType="Log_Protocol_Controller"/>
      <causes>
        <port id="rcvRes" specType="Log_Handler"/>
      </causes>
    </accept>
    <accept>
      <effect id="rcvFromDri" specType="Log_Protocol_Controller"/>
      <causes>
        <port id="send_log" specType="Log_Handler"/>
      </causes>
    </accept>
    <accept>
      <effect id="send" specType="Log_Protocol_Controller"/>
```

```
    <causes>
      <port id="add" specType="AP_ReceiverFifo"/>
    </causes>
  </accept>
  <accept>
    <effect id="rcvDriver" specType="Log_Protocol_Controller"/>
    <causes>
      <port id="send_log" specType="Log_Handler"/>
    </causes>
  </accept>
  <accept>
    <effect id="receive" specType="Log_Protocol_Controller"/>
    <causes>
      <port id="sndToController" specType="/f/e/Z"/>
    </causes>
  </accept>
  <accept>
    <effect id="add" specType="AP_ReceiverFifo"/>
    <causes>
      <port id="send" specType="Control_Protocol_Controller"/>
      <port id="send" specType="Camera_Protocol_Controller"/>
      <port id="send" specType="Log_Protocol_Controller"/>
      <port id="send" specType="Power_Supply_Protocol_Controller"/>
    </causes>
  </accept>
  <accept>
    <effect id="rm" specType="AP_ReceiverFifo"/>
    <causes>
      <port id="receive" specType="AP_MessageWorker"/>
    </causes>
  </accept>
  <accept>
    <effect id="sndRes" specType="AP_MessageWorker"/>
    <causes>
      <port id="getReq" specType="/f/e/Z"/>
    </causes>
  </accept>
  <accept>
    <effect id="receive" specType="AP_MessageWorker"/>
    <causes>
      <port id="rm" specType="AP_ReceiverFifo"/>
    </causes>
  </accept>
  <accept>
    <effect id="sndReq" specType="AP_MessageWorker"/>
    <causes>
      <port id="getRes" specType="AP_ResponseWorker"/>
    </causes>
  </accept>
  <accept>
    <effect id="sndToController" specType="AP_RequestWorker"/>
    <causes>
      <port id="receive" specType="Control_Protocol_Controller"/>
      <port id="receive" specType="Log_Protocol_Controller"/>
      <port id="receive" specType="Power_Supply_Protocol_Controller"/>
      <port id="receive" specType="Camera_Protocol_Controller"/>
    </causes>
  </accept>
  <accept>
    <effect id="getReq" specType="AP_RequestWorker"/>
    <causes>
      <port id="sndRes" specType="AP_MessageWorker"/>
    </causes>
  </accept>
  <accept>
    <effect id="getRes" specType="AP_ResponseWorker"/>
    <causes>
      <port id="sndReq" specType="AP_MessageWorker"/>
    </causes>
  </accept>
```

```
<accept>
  <effect id="sndToDri" specType="Control_Protocol_Controller"/>
  <causes>
    <port id="rcvReq" specType="Control_Disconnect_Handler"/>
    <port id="rcvReq" specType="Control_Connect_Handler"/>
  </causes>
</accept>
<accept>
  <effect id="rcvFromDri" specType="Control_Protocol_Controller"/>
  <causes>
    <port id="sndRes" specType="Control_Connect_Handler"/>
    <port id="sndRes" specType="Control_Disconnect_Handler"/>
  </causes>
</accept>
<accept>
  <effect id="send" specType="Control_Protocol_Controller"/>
  <causes>
    <port id="add" specType="AP_ReceiverFifo"/>
  </causes>
</accept>
<accept>
  <effect id="rcvDriver" specType="Control_Protocol_Controller"/>
  <causes>
    <port id="sndRes" specType="Control_Connect_Handler"/>
    <port id="sndRes" specType="Control_Disconnect_Handler"/>
  </causes>
</accept>
<accept>
  <effect id="receive" specType="Control_Protocol_Controller"/>
  <causes>
    <port id="sndToController" specType="AP_RequestWorker"/>
  </causes>
</accept>
<accept>
  <effect id="sndToDri" specType="Power_Supply_Protocol_Controller"/>
  <causes>
    <port id="rcvReq" specType="Power_Supply_Handler"/>
  </causes>
</accept>
<accept>
  <effect id="rcvFromDri" specType="Power_Supply_Protocol_Controller"/>
  <causes>
    <port id="sndRes" specType="Power_Supply_Handler"/>
  </causes>
</accept>
<accept>
  <effect id="send" specType="Power_Supply_Protocol_Controller"/>
  <causes>
    <port id="add" specType="AP_ReceiverFifo"/>
  </causes>
</accept>
<accept>
  <effect id="rcvDriver" specType="Power_Supply_Protocol_Controller"/>
  <causes>
    <port id="sndRes" specType="Power_Supply_Handler"/>
  </causes>
</accept>
<accept>
  <effect id="receive" specType="Power_Supply_Protocol_Controller"/>
  <causes>
    <port id="sndToController" specType="AP_RequestWorker"/>
  </causes>
</accept>
<accept>
  <effect id="sndToDri" specType="Camera_Protocol_Controller"/>
  <causes>
    <port id="rcvReq" specType="Camera_Stream_Handler"/>
    <port id="rcvReq" specType="Camera_Capture_Handler"/>
  </causes>
</accept>
```

```
<accept>
  <effect id="rcvFromDri" specType="Camera_Protocol_Controller"/>
  <causes>
    <port id="sndRes" specType="Camera_Capture_Handler"/>
    <port id="sndRes" specType="Camera_Stream_Handler"/>
  </causes>
</accept>
<accept>
  <effect id="send" specType="Camera_Protocol_Controller"/>
  <causes>
    <port id="add" specType="AP_ReceiverFifo"/>
  </causes>
</accept>
<accept>
  <effect id="rcvDriver" specType="Camera_Protocol_Controller"/>
  <causes>
    <port id="sndRes" specType="Camera_Capture_Handler"/>
    <port id="sndRes" specType="Camera_Stream_Handler"/>
  </causes>
</accept>
<accept>
  <effect id="receive" specType="Camera_Protocol_Controller"/>
  <causes>
    <port id="sndToController" specType="AP_RequestWorker"/>
  </causes>
</accept>
<accept>
  <effect id="rcvReq" specType="Power_Supply_Handler"/>
  <causes>
    <port id="sndToDri" specType="Power_Supply_Protocol_Controller"/>
  </causes>
</accept>
<accept>
  <effect id="sndRes" specType="Power_Supply_Handler"/>
  <causes>
    <port id="rcvDriver" specType="Power_Supply_Protocol_Controller"/>
    <port id="rcvFromDri" specType="Power_Supply_Protocol_Controller"/>
  </causes>
</accept>
<accept>
  <effect id="rcvRes" specType="Log_Handler"/>
  <causes>
    <port id="sndToDri" specType="Log_Protocol_Controller"/>
  </causes>
</accept>
<accept>
  <effect id="send_log" specType="Log_Handler"/>
  <causes>
    <port id="rcvDriver" specType="Log_Protocol_Controller"/>
    <port id="rcvFromDri" specType="Log_Protocol_Controller"/>
  </causes>
</accept>
<accept>
  <effect id="rcvReq" specType="Control_Connect_Handler"/>
  <causes>
    <port id="sndToDri" specType="Control_Protocol_Controller"/>
  </causes>
</accept>
<accept>
  <effect id="sndRes" specType="Control_Connect_Handler"/>
  <causes>
    <port id="rcvFromDri" specType="Control_Protocol_Controller"/>
  </causes>
</accept>
<accept>
  <effect id="rcvReq" specType="Camera_Stream_Handler"/>
  <causes>
    <port id="sndToDri" specType="Camera_Protocol_Controller"/>
  </causes>
</accept>
```

```
    <accept>
      <effect id="sndRes" specType="Camera_Stream_Handler"/>
      <causes>
        <port id="rcvDriver" specType="Camera_Protocol_Controller"/>
        <port id="rcvFromDri" specType="Camera_Protocol_Controller"/>
      </causes>
    </accept>
    <accept>
      <effect id="rcvReq" specType="Control_Disconnect_Handler"/>
      <causes>
        <port id="sndToDri" specType="Control_Protocol_Controller"/>
      </causes>
    </accept>
    <accept>
      <effect id="sndRes" specType="Control_Disconnect_Handler"/>
      <causes>
        <port id="rcvDriver" specType="Control_Protocol_Controller"/>
        <port id="rcvFromDri" specType="Control_Protocol_Controller"/>
      </causes>
    </accept>
    <accept>
      <effect id="rcvReq" specType="Camera_Capture_Handler"/>
      <causes>
        <port id="sndToDri" specType="Camera_Protocol_Controller"/>
      </causes>
    </accept>
    <accept>
      <effect id="sndRes" specType="Camera_Capture_Handler"/>
      <causes>
        <port id="rcvDriver" specType="Camera_Protocol_Controller"/>
        <port id="rcvFromDri" specType="Camera_Protocol_Controller"/>
      </causes>
    </accept>
  </accepts>
  <requires>
    <require>
      <effect id="sndToDri" specType="Log_Protocol_Controller"/>
      <causes>
        <option>
          <causes>
            <port id="rcvRes" specType="Log_Handler"/>
          </causes>
        </option>
      </causes>
    </require>
    <require>
      <effect id="rcvFromDri" specType="Log_Protocol_Controller"/>
      <causes>
        <option>
          <causes>
            <port id="send_log" specType="Log_Handler"/>
          </causes>
        </option>
      </causes>
    </require>
    <require>
      <effect id="send" specType="Log_Protocol_Controller"/>
      <causes>
        <option>
          <causes>
            <port id="add" specType="AP_ReceiverFifo"/>
          </causes>
        </option>
      </causes>
    </require>
    <require>
      <effect id="rcvDriver" specType="Log_Protocol_Controller"/>
      <causes>
        <option>
          <causes>
```

```
          <port id="send_log" specType="Log_Handler"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="receive" specType="Log_Protocol_Controller"/>
    <causes>
      <option>
        <causes>
          <port id="sndToController" specType="AP_RequestWorker"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="add" specType="AP_ReceiverFifo"/>
    <causes>
      <option>
        <causes>
          <port id="send" specType="Control_Protocol_Controller"/>
        </causes>
      </option>
      <option>
        <causes>
          <port id="send" specType="Camera_Protocol_Controller"/>
        </causes>
      </option>
      <option>
        <causes>
          <port id="send" specType="Log_Protocol_Controller"/>
        </causes>
      </option>
      <option>
        <causes>
          <port id="send" specType="Power_Supply_Protocol_Controller"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="rm" specType="AP_ReceiverFifo"/>
    <causes>
      <option>
        <causes>
          <port id="receive" specType="AP_MessageWorker"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="sndRes" specType="AP_MessageWorker"/>
    <causes>
      <option>
        <causes>
          <port id="getReq" specType="AP_RequestWorker"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="receive" specType="AP_MessageWorker"/>
    <causes>
      <option>
        <causes>
          <port id="rm" specType="AP_ReceiverFifo"/>
        </causes>
      </option>
    </causes>
  </require>
```

```
<require>
  <effect id="sndReq" specType="AP_MessageWorker"/>
  <causes>
    <option>
      <causes>
        <port id="getRes" specType="AP_ResponseWorker"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="sndToController" specType="AP_RequestWorker"/>
  <causes>
    <option>
      <causes>
        <port id="receive" specType="Control_Protocol_Controller"/>
      </causes>
    </option>
    <option>
      <causes>
        <port id="receive" specType="Log_Protocol_Controller"/>
      </causes>
    </option>
    <option>
      <causes>
        <port id="receive" specType="Power_Supply_Protocol_Controller"/>
      </causes>
    </option>
    <option>
      <causes>
        <port id="receive" specType="Camera_Protocol_Controller"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="getReq" specType="AP_RequestWorker"/>
  <causes>
    <option>
      <causes>
        <port id="sndRes" specType="AP_MessageWorker"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="getRes" specType="AP_ResponseWorker"/>
  <causes>
    <option>
      <causes>
        <port id="sndReq" specType="AP_MessageWorker"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="sndToDri" specType="Control_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="rcvReq" specType="Control_Disconnect_Handler"/>
      </causes>
    </option>
    <option>
      <causes>
        <port id="rcvReq" specType="Control_Connect_Handler"/>
      </causes>
    </option>
  </causes>
</require>
```

```
<require>
  <effect id="rcvFromDri" specType="Control_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="sndRes" specType="Control_Connect_Handler"/>
      </causes>
    </option>
    <option>
      <causes>
        <port id="sndRes" specType="Control_Disconnect_Handler"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="send" specType="Control_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="add" specType="AP_ReceiverFifo"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="rcvDriver" specType="Control_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="sndRes" specType="Control_Connect_Handler"/>
      </causes>
    </option>
    <option>
      <causes>
        <port id="sndRes" specType="Control_Disconnect_Handler"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="receive" specType="Control_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="sndToController" specType="AP_RequestWorker"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="sndToDri" specType="Power_Supply_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="rcvReq" specType="Power_Supply_Handler"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="rcvFromDri" specType="Power_Supply_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="sndRes" specType="Power_Supply_Handler"/>
        <port id="sndRes" specType="Power_Supply_Handler"/>
      </causes>
    </option>
  </causes>
```

```
</require>
<require>
  <effect id="send" specType="Power_Supply_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="add" specType="AP_ReceiverFifo"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="rcvDriver" specType="Power_Supply_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="sndRes" specType="Power_Supply_Handler"/>
        <port id="sndRes" specType="Power_Supply_Handler"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="receive" specType="Power_Supply_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="sndToController" specType="AP_RequestWorker"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="sndToDri" specType="Camera_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="rcvReq" specType="Camera_Stream_Handler"/>
      </causes>
    </option>
    <option>
      <causes>
        <port id="rcvReq" specType="Camera_Capture_Handler"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="rcvFromDri" specType="Camera_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="sndRes" specType="Camera_Capture_Handler"/>
      </causes>
    </option>
    <option>
      <causes>
        <port id="sndRes" specType="Camera_Stream_Handler"/>
      </causes>
    </option>
  </causes>
</require>
<require>
  <effect id="send" specType="Camera_Protocol_Controller"/>
  <causes>
    <option>
      <causes>
        <port id="add" specType="AP_ReceiverFifo"/>
      </causes>
    </option>
```

```
    </causes>
  </require>
  <require>
    <effect id="rcvDriver" specType="Camera_Protocol_Controller"/>
    <causes>
      <option>
        <causes>
          <port id="sndRes" specType="Camera_Capture_Handler"/>
        </causes>
      </option>
      <option>
        <causes>
          <port id="sndRes" specType="Camera_Stream_Handler"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="receive" specType="Camera_Protocol_Controller"/>
    <causes>
      <option>
        <causes>
          <port id="sndToController" specType="AP_RequestWorker"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="rcvReq" specType="Power_Supply_Handler"/>
    <causes>
      <option>
        <causes>
          <port id="sndToDri" specType="Power_Supply_Protocol_Controller"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="sndRes" specType="Power_Supply_Handler"/>
    <causes>
      <option>
        <causes>
          <port id="rcvDriver" specType="Power_Supply_Protocol_Controller"/>
        </causes>
      </option>
      <option>
        <causes>
          <port id="rcvFromDri" specType="Power_Supply_Protocol_Controller"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="rcvRes" specType="Log_Handler"/>
    <causes>
      <option>
        <causes>
          <port id="sndToDri" specType="Log_Protocol_Controller"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="send_log" specType="Log_Handler"/>
    <causes>
      <option>
        <causes>
          <port id="rcvDriver" specType="Log_Protocol_Controller"/>
        </causes>
      </option>
```

```
      <option>
        <causes>
          <port id="rcvFromDri" specType="Log_Protocol_Controller"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="rcvReq" specType="Control_Connect_Handler"/>
    <causes>
      <option>
        <causes>
          <port id="sndToDri" specType="Control_Protocol_Controller"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="sndRes" specType="Control_Connect_Handler"/>
    <causes>
      <option>
        <causes>
          <port id="rcvFromDri" specType="Control_Protocol_Controller"/>
        </causes>
      </option>
      <option>
        <causes>
          <port id="rcvFromDri" specType="Control_Protocol_Controller"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="rcvReq" specType="Camera_Stream_Handler"/>
    <causes>
      <option>
        <causes>
          <port id="sndToDri" specType="Camera_Protocol_Controller"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="sndRes" specType="Camera_Stream_Handler"/>
    <causes>
      <option>
        <causes>
          <port id="rcvDriver" specType="Camera_Protocol_Controller"/>
        </causes>
      </option>
      <option>
        <causes>
          <port id="rcvFromDri" specType="Camera_Protocol_Controller"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="rcvReq" specType="Control_Disconnect_Handler"/>
    <causes>
      <option>
        <causes>
          <port id="sndToDri" specType="Control_Protocol_Controller"/>
        </causes>
      </option>
    </causes>
  </require>
  <require>
    <effect id="sndRes" specType="Control_Disconnect_Handler"/>
    <causes>
```

```
        <option>
          <causes>
            <port id="rcvDriver" specType="Control_Protocol_Controller"/>
          </causes>
        </option>
        <option>
          <causes>
            <port id="rcvFromDri" specType="Control_Protocol_Controller"/>
          </causes>
        </option>
      </causes>
    </require>
    <require>
      <effect id="rcvReq" specType="Camera_Capture_Handler"/>
      <causes>
        <option>
          <causes>
            <port id="sndToDri" specType="Camera_Protocol_Controller"/>
          </causes>
        </option>
      </causes>
    </require>
    <require>
      <effect id="sndRes" specType="Camera_Capture_Handler"/>
      <causes>
        <option>
          <causes>
            <port id="rcvDriver" specType="Camera_Protocol_Controller"/>
          </causes>
        </option>
        <option>
          <causes>
            <port id="rcvFromDri" specType="Camera_Protocol_Controller"/>
          </causes>
        </option>
      </causes>
    </require>
  </requires>
</glue>
```