DesignBIP: A Design Studio for Modeling and Generating Systems with BIP

Anastasia Mavridou
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
anastasia.mavridou@vanderbilt.edu

Joseph Sifakis
Verimag-Bâtiment IMAG
Universitè Grenoble Alpes
38401 St Martin d'Hères, France
Joseph.Sifakis@imag.fr

Janos Sztipanovits
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
janos.sztipanovits@vanderbilt.edu

The Behavior-Interaction-Priority (BIP) framework — rooted in rigorous semantics — allows the construction of systems that are correct-by-design. BIP has been effectively used for the construction and analysis of large systems such as robot controllers and satellite on-board software. Nevertheless, the specification of BIP models is done in a purely textual manner without any code editor support. To facilitate the specification of BIP models, we present DesignBIP, a web-based, collaborative, version-controlled design studio. To promote model scaling and reusability of BIP models, we use a graphical language for modeling parameterized BIP models with rigorous semantics. We present the various services provided by the design studio, including model editors, code editors, consistency checking mechanisms, code generators, and integration with the JavaBIP tool-set.

1 Introduction

Modeling languages are often used for designing complex systems. Using dedicated design studios allows increasing the understandability and usability of modeling languages, as well as decreasing development costs by eliminating errors at design time. Design studio components can be organized in the following three categories: 1) *semantic integration*, 2) *service integration*, and 3) *tool integration*. Semantic integration components comprise the domain of the modeling language, i.e., its *metamodel* that explicitly specifies the building blocks of the language and their relations. Service integration components include dedicated model editors, code editors, and GUI/Visualization components for modeling and simulating results. Additionally, service integration components include model transformation and code generation services, model repositories, and version control services. Finally, tool integration components consist in integrated tools such as run-times and verification tools.

Figure 1 shows the main steps of the workflow of a design studio. Initially, models are designed using dedicated model editors. Optionally, design patterns stored in model repositories may be used to simplify the modeling process. Next, the checking loop starts (step 1), where the models are checked for conformance. If the required conformance conditions are not satisfied by the model, the checking mechanism must point back to the problematic nodes of the model in the model editor and inform the developer of the inconsistency causes to facilitate model refinement. Finally, when the conformance conditions are satisfied (step 2), the refined models may be analyzed and/or executed (step 3) by using

Submitted to:
MeTRiD 2018

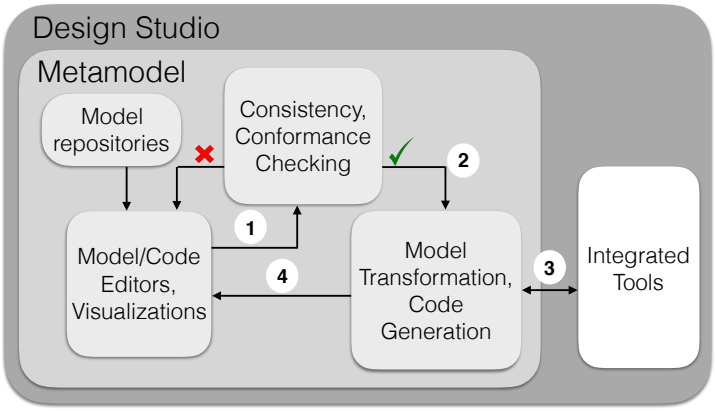


Figure 1: Main design studio components and work-flow

integrated, into the design studio, third party tools. The output of the tools is then collected and sent back to the model editors (step 4) for visualization of analysis or execution results.

We present the DesignBIP studio for modeling and generating systems with the BIP [2] framework. BIP comprises a language with rigorous operational semantics and a dedicated tool-set including code generators, run-time support tools, i.e., BIP engines, and verification tools [5,7]. Depending on the application domain, BIP offers several compilation chains, targeting different execution platforms and programming languages such as C++ [2] Java [8] Haskell, and Scala [15].

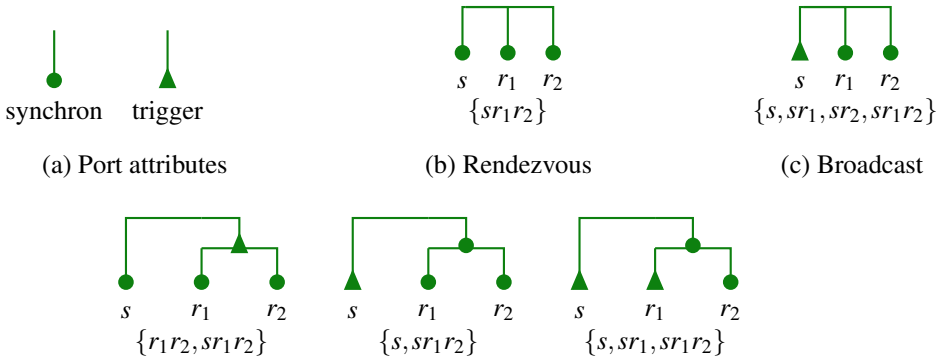
The specification of models in the BIP framework is done by using the BIP language in a textual manner [1] without offering any dedicated code editors. Thus, developing large systems with the BIP toolset can be challenging and error prone. In DesignBIP we have opted for a graphical language to enhance readability and easiness of expression. DesignBIP offers a complete modeling solution, in which we have integrated the tools offered by JavaBIP, the Java-based implementation of BIP [8]. Relying on the observation that systems are usually built from multiple instances of the same component type, we propose a parameterized graphical language for BIP that enhances scalability and reduces the model size.

DesignBIP is a web-based, collaborative, version controlled design studio based on WebGME [22]. DesignBIP allows real-time collaboration between multiple developers. Project changes are committed and versioned, which enables branching, merging and viewing the history of a project. DesignBIP is easily accessed through a web interface and is open source[1]. Our contributions are as follows:

- We extend *architecture diagrams* [23], a graphical parameterized language, to accommodate the specification of BIP parameterized models.
- We prove a set of necessary and sufficient conditions for checking the encodability of parameterized BIP graphical models into logical formulas.
- We study the model transformation from graphical models to logical formulas and develop code generation plugins.
- We develop dedicated BIP model editors, code editors, and model repositories.
- We integrate the JavaBIP engine and provide visualization of its output.

Paper organization: Section 2 describes the BIP language. Section 3 describes the parameterized graphical language of DesignBIP. Section 4 describes service integration components, i.e., model and code editors, model repositories, and code generators. Section 5 describes the integration with the JavaBIP engine. Section 6 discusses related work. Section 7 discusses concluding remarks and future work.

[1] https://github.com/DesignBIP/DesignBIP



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

(d) Hierarchical connectors

Figure 2: BIP connectors and their associated interaction sets

2 The BIP Language

Component behavior in BIP is described by Labelled Transition Systems (LTS). LTS transitions are of three types: *enforceable*, *spontaneous*, and *internal*[2]. Enforceable transitions are handled by the BIP-engine and are labeled with *ports*. Ports form the interface of a component and are used to define interactions with other components. Spontaneous transitions take into account changes in the environment and, thus, they are not handled by the BIP-engine but rather executed after detection of external events. Finally, internal transitions allow a component to update its state based on internal information.

An *interaction* in BIP is a non-empty set of ports that defines allowed synchronization of actions among components. BIP interactions represent a clean, abstract concept of *architecture* which is separated from component behavior. Interaction models can be represented in many equivalent ways. Among these are connectors [9] and Boolean formulas [10] on variables representing port participation in interactions. Connectors are most appropriate for graphical design and interaction representation, whereas Boolean formulas are most appropriate for efficient encoding and manipulation by the BIP-engine.

BIP connectors contain ports, which form their interface. Each port of a connector has an attribute *trigger* (represented by a triangle, Figure 2a) or *synchron* (represented by a bullet, Figure 2a). Given a connector involving a set of ports $\{p_1,...,p_n\}$, the set of its interactions is defined as follows: an interaction is any non-empty subset of $\{p_1,...,p_n\}$ which contains some port that is a trigger (Figure 2c); otherwise, (if all ports are synchrons) the only possible interaction is the maximal one that is, $\{p_1,...,p_n\}$ (Figure 2b). The same principle is recursively extended to hierarchical connectors, where one interaction from each subconnector is used to form an allowed interaction according to the synchron/trigger typing of the connector nodes (Figure 2d).

Alternatively, *interaction logic* can be used to define interaction models. The propositional interaction logic (PIL) is defined by the grammar:

$$\phi ::= true \mid p \mid \overline{\phi} \mid \phi \vee \phi\,,$$

with any $p \in P$, where P is the set of ports of a BIP system. Conjunction is defined as follows: $\phi_1 \wedge \phi_2 \stackrel{def}{=} \overline{(\overline{\phi_1} \vee \overline{\phi_2})}$. To simplify notation, we omit conjunction in monomials, e.g., writing sr_1r_2 instead of $s \wedge r_1 \wedge r_2$. Let γ be a non-empty set of interactions. The meaning of a PIL formula ϕ is defined by the

[2]JavaBIP includes all three, whereas BIP1 and BIP2 include the enforceable and internal types.

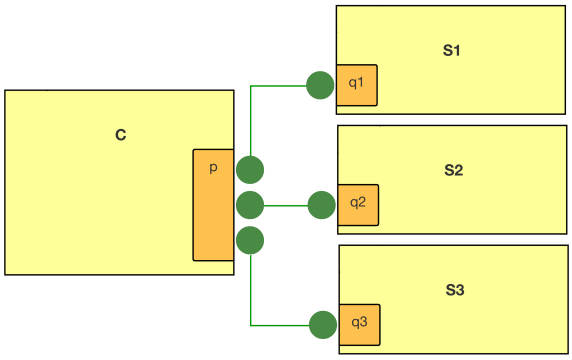


Figure 3: A Star architecture

satisfaction relation: $\gamma \models \phi$ 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 $p \notin a$.

Consider the *Star architecture* shown in Figure 3, 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 rendezvous connectors. Component C has a single port p and all other components have a single port q_i $(i = 1,2,3)$. The corresponding PIL formula is: $pq_1\overline{q_2}\,\overline{q_3} \vee p\overline{q_1}\,q_2\overline{q_3} \vee p\overline{q_1}\,\overline{q_2}\,q_3$.

To define interactions independently from the number of component instances, PIL can be extended with quantification over components [11]. This extension is particularly useful because, in practice, systems are built from multiple component instances of the same component type. Similarly to [11], JavaBIP uses a macro-notation based on FOIL that includes two macros.

The **Require** macro defines ports required for interaction. Let $T_1, T_2 \in \mathscr{T}$ be two component types. For instance:

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

means that, to participate in an interaction, each of the ports p of component instances of type T_1 requires either the participation of *precisely two* of the ports q of component instances of type T_2 or one instance of r. Notice the semicolon in the macro that separates the two options.

The **Accept** macro defines optional ports for participation, i.e., it defines the boundary of interactions. This is expressed by explicitly excluding from interactions all the ports that are not optional. For instance, if p,q,r is the set of port types of component types $T_1, T_2 \in \mathscr{T}$ then:

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

means that instances of r are excluded from interaction with instances of p. To illustrate the use of the macros, let us define the Star architecture style with Require/Accept:

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

The syntax and semantics of first-order interaction logic (FOIL) as well as the Require/Accept macronotation are presented in greater detail in Appendix A.

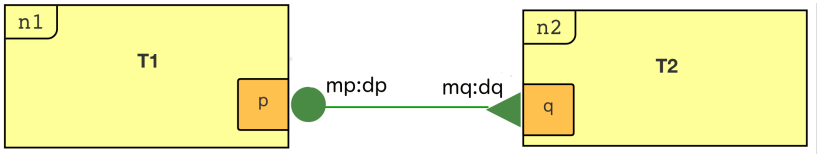


Figure 4: A BIP architecture diagram

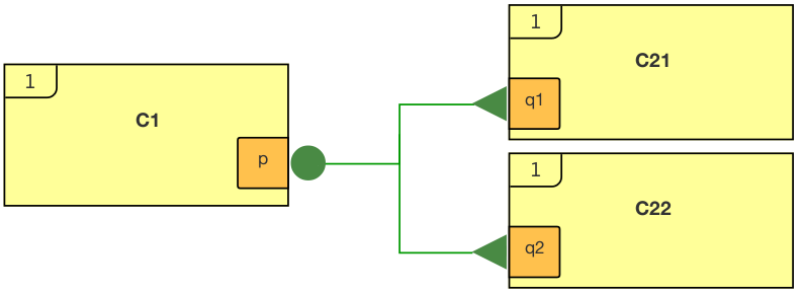


Figure 5: A conforming architecture to the diagram in Figure 4

3 Semantic Integration Components

We present the BIP parameterized graphical language that was integrated in DesignBIP. The DesignBIP metamodel can be found in Appendix B.

3.1 Architecture Diagrams for BIP

Architecture diagrams [23] is a parameterized graphical language for the description of the structure of a system by showing the system's component types and their attributes for coordination.We extend the definition of architecture diagrams with triggers and synchrons to define BIP connectors. A BIP architecture diagram consists of a set of *component types* and a set of *connector motifs*. Each component type T is characterized by a set of *port types* $T.P$ and a *cardinality* parameter n, which specifies the number of instances of T. Figure 4 shows an architecture diagram consisting of two component types T_1 and T_2 with n_1 and n_2 instances and port types p and q, respectively. Instantiated components have port instances p_i, q_j for i, j belonging to the intervals $[1,n_1]$, $[1,n_2]$, respectively.

Connector motifs are non-empty sets of port types. Each port type p in a connector motif has two constraints represented as a pair $m : d$. Multiplicity m of a port type constrains the number of port instances of this type that are involved in each connector defined by the connector motif. Degree d of a port type constrains the number of connectors attached to every port instance of this type. Additionally, each port type has a typing (attribute) represented by t_p, which can be either *trigger* (represented by a triangle) or *synchron* (represented by a bullet) (see BIP connectors in Section 2). A connector motif defines a set of possible configurations, where a configuration is a non-empty set of connectors. The meaning of a diagram is the union of all configurations corresponding to each connector motif of the diagram. Let us present the semantics of connector motifs through the example of Figure 4, which has a single connector motif involving port types p and q.

Figure 5 shows the unique configuration obtained from the diagram of Figure 4 by taking $n_1 = 1$, $m_p = 1$, $d_p = 1$; $n_2 = 2$, $m_q = 2$ and $d_q = 1$. This is the result of composition of constraints for port types p and q. For instance, since the multiplicity of q is 2, then both q_1 and q_2 must be involved in the

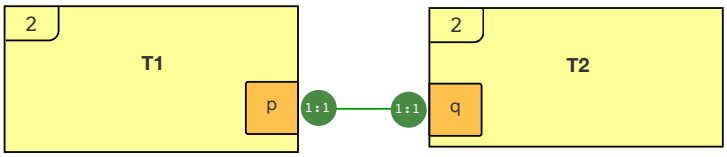


Figure 6: An architecture diagram that cannot be encoded into FOIL

the same connector. The degrees of p and q are equal to 1, thus there is exactly one connector attached to their port instances. Port instances retain the typing of their corresponding port types. The set of interactions defined by the connector in Figure 5 is the following: $\{q_1, q_2, q_1q_2, q_1p, q_2p, q_1q_2p\}$.

Formally, a *BIP architecture diagram* $\mathscr{D} = \langle \mathscr{T}, \mathscr{C} \rangle$ consists of:

- a set of *component types* $\mathscr{T} = \{T_1, \ldots, T_k\}$ of the form $T = (T.P, n)$, where $T.P \neq \emptyset$ is the set of *port types* of component type T and $n \in \mathbb{N}$ is the *cardinality* parameter associated to component type T
- a set of *connector motifs* $\mathscr{C} = \{\Gamma_1, \ldots, \Gamma_l\}$ of the form $\Gamma = (a, \{m_p : d_p,\ t_p\}_{p \in a})$, where
  - $\emptyset \neq a \subset \bigcup_{i=1}^{k} T_i.P$ is a set of port types
  - $m_p, d_p \in \mathbb{N}$ (with $m_p > 0$) are the *multiplicity* and *degree* associated to port type $p \in a$
  - $t_p \in \{synchron,\ trigger\}$ is the typing of port type $p \in a$

For a component $c \in \mathscr{B}$ and a component type T, we say that c *is of type* T if the ports of c are in a bijective correspondence with the port types in T.

An architecture $\langle \mathscr{B}, \gamma \rangle$ *conforms* to a diagram $\langle \mathscr{T}, \mathscr{C} \rangle$ if, for each $i \in [1,k]$, the number of components of type T_i in $\mathscr{B}$ is equal to n_i and γ can be partitioned into disjoint sets $\gamma_1, \ldots, \gamma_l$, such that, for each connector motif $\Gamma_i = (a, \{m_p : d_p,\ t_p\}_{p \in a}) \in \mathscr{C}$ and each $p \in a$,

1. in each connector in γ_i there are exactly m_p instances of p typed as t_p,
2. each instance of p is involved in exactly d_p connectors in γ_i

The meaning of a BIP architecture diagram is the set of all architectures that conform to it.

3.1.1 Conformance Conditions

DesignBIP encodes connector motifs in the Require/Accept macronotation (Section 2) in order to give the latter as input to the integrated JavaBIP-engine. Nevertheless, the semantic domains of BIP architecture diagrams and interaction logic (Section 2) do not coincide. An architecture diagram defines a set of configurations, whereas, an interaction logic formula defines exactly one configuration. Consider the architecture diagram shown in Figure 6 with a single connector motif, which defines two configurations: $\gamma_1 = \{p_1q_1, p_2q_2\}$ and $\gamma_2 = \{p_1q_2, p_2q_1\}$, and thus, cannot be encoded into interaction logic. Let us now consider the architecture diagram shown in Figure 7, which is a variation of the diagram shown in Figure 6 with degrees set to $d_p = d_q = 2$. This diagram defines exactly one configuration and thus, can be encoded into interaction logic. In particular, it defines the configuration $\gamma = \{p_1q_1, p_2q_2, p_1q_2, p_2q_1\}$. This shows that we can restrict an architecture diagram to define exactly one configuration by constraining its multiplicities or degrees.

We denote $s_p = n_p \cdot d_p / m_p \in \mathbb{N}$ the *matching factor* of a port type p, where n_p is the cardinality of the component type that contains p. The matching factors of all port types participating in the same connector motif must be equal integers, in which case they represent the number of connectors defined

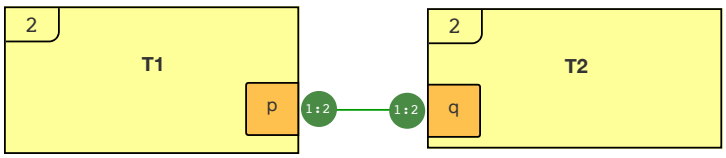


Figure 7: An architecture diagram that can be encoded into FOIL

by the connector motif. The maximum number of distinct connectors defined by a connector motif $\Gamma = (a, \{m_p : d_p,\ t_p\}_{p \in a})$ is equal to $\prod_{q \in a} \binom{n_q}{m_q}$. Consider the connector motif shown in Figure 6. The matching factors of its port types are $s_p = s_q = 2$ and are not equal to $\binom{2}{1} \cdot \binom{2}{1}$, which represents the maximum number of connectors that can be defined by this connector motif. The matching factors of the connector motif shown in Figure 7 is 4.

Proposition 3.1 provides the necessary and sufficient conditions for a BIP diagram to define exactly one conforming architecture for each evaluation of its cardinality parameters. If these conditions hold, then the diagram can be encoded into FOIL. The encoding conditions are as follows: 1) the multiplicity of a port type must be less than or equal to the number of component instances that contain this port and 2) the matching factors of all port types participating in the same connector motif must be equal to the maximum number of connectors that the connector motif defines. Since, by the semantics of diagrams, connector motifs correspond to disjoint sets of connectors, these conditions are applied separately to each connector motif. The proof of Proposition 3.1 can be found in Appendix C. Corollary 3.2 follows directly from Proposition 3.1.

Proposition 3.1. *A BIP architecture diagram has exactly one conforming architecture iff, for each connector motif* $\Gamma = (a, \{m_p : d_p,\ t_p\}_{p \in a})$ *and each* $p \in a$, *we have 1)* $m_p \leq n_p$ *and 2)* $s_p = \prod_{q \in a} \binom{n_q}{m_q}$.

Corollary 3.2. *A BIP architecture diagram can be specified in FOIL using the Require/Accept macro-notation iff, for each connector motif* $\Gamma = (a, \{m_p : d_p,\ t_p\}_{p \in a})$ *and each* $p \in a$, *we have 1)* $m_p \leq n_p$ *and 2)* $s_p = \prod_{q \in a} \binom{n_q}{m_q}$.

4 Service Integration Components

We present the model and code editors, the code generators, and the model repositories of DesignBIP.

4.1 Model and Code editors

A developer provides the system specification by using the dedicated model and Java code editors of DesignBIP. In particular, the developer must specify 1) component behavior in the form of BIP LTS, 2) component interaction in the form of BIP architecture diagrams and 3) the actions associated with transitions and guards, as well as variable declarations directly in Java. Figures 8 and 9 present the DesignBIP LTS and BIP diagram model editors, as well as the Java code editor. In the code editor, the darker parts represent code that was automatically generated by the input given in the model editors, while the bright code parts represent input given directly in the code editor. In the LTS model editor, enforceable and internal transitions are illustrated with solid arrows, while spontaneous transitions are illustrated with dashed arrows. The code and model editors are tightly synchronized, i.e., changes are instantaneously propagated.

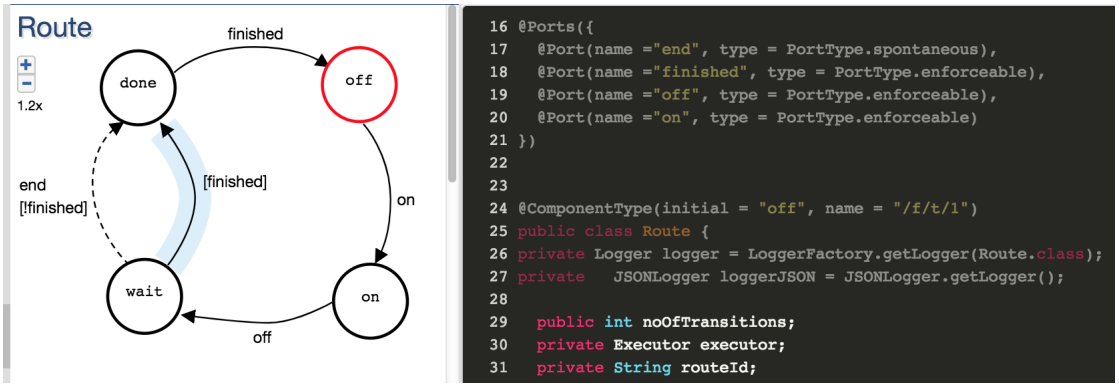


Figure 8: DesignBIP LTS model editor and Java code editor

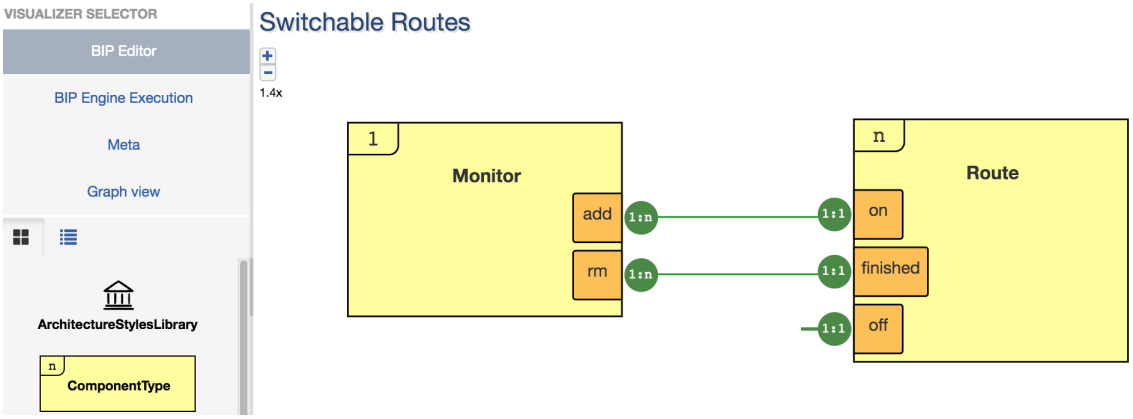


Figure 9: DesignBIP architecture diagrams model editor

4.2 Behavior generation plugin: LTS to Java code

As explained earlier, a developer graphically specifies the LTS that represents component behavior. DesignBIP generates the Java code that describes the LTS specified by the developer. In particular, DesignBIP generates code in the form of Java annotations that describes the ports, component types, transitions, and guards of each LTS. For instance, Java annotations describing the ports of the `Route` component type (Figure 9) are shown in the right-hand side of Figure 8.

Firstly, before the plugin generates the Java code, it checks the correct instantiation of each specified LTS according to the constraints defined in the DesignBIP metamodel (Appendix B). For instance, the plugin checks whether each LTS has exactly one initial state. If errors exist, DesignBIP returns to the developer a message explaining the error and pointers (displayed as `Show node`) to the incorrect nodes of the specified model as shown in Figure 10. In the case of a correct behavioral model, DesignBIP returns a set of Java files, i.e., one Java file for each specified component type. The complete generated Java annotations for the `Route` component type are presented in Appendix D.

4.3 Interaction generation plugin: BIP architecture diagrams to XML code

We propose Algorithm 1, with polynomial-time complexity for the encoding of a BIP architecture diagram into Require/Accept macros (Section 2). For each port type, we instantiate two sets of variables:

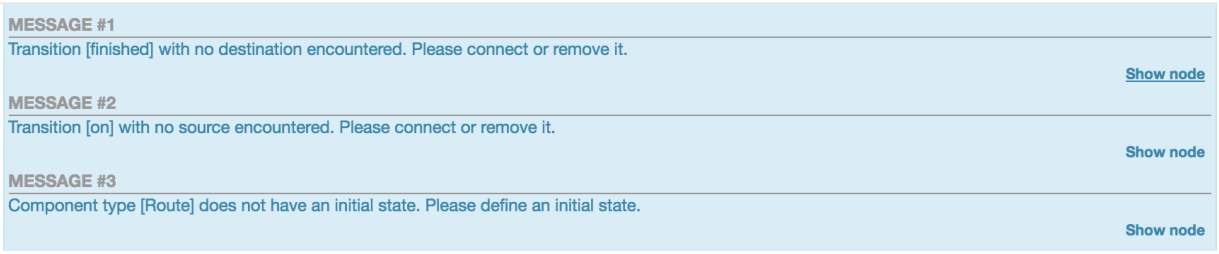


Figure 10: Behavioral errors as returned by DesignBIP

`require` and `accept`. For the sake of simplicity, we write p instead of $T.p$.

The `accept` set of p contains the right hand side of **Accept** and is constructed as follows. For each connector motif attached to p, if its size is: 1) equal to 1, i.e., singleton connector motif, then we add - in `accept`[3]; 2) greater than 1 and the multiplicity of p is greater than 1, we add in `accept` all port types of the connector motif including p; 3) greater than 1 and the multiplicity of p is equal to 1, we add all port types of the connector motif except for p to `accept`.

The `require` set of p contains the right hand side of **Require** and is constructed as follows. For each connector motif attached to p, if its size is: 1) equal to 1 or p is typed as trigger then we add - to `require`[4]; 2) greater than 1 and there exists at least one trigger, we add to `require` as many options as the number of triggers. In each option we add a trigger; 3) greater than 1 and there are no triggers, we add to `require` all port types of the connector motif except for p as many times as their associated multiplicity and $m_p - 1$ times the port type p, to form a single option.

Before generating the XML code, DesignBIP checks the conformance conditions presented in Section ref:conformance. Additionally, DesignBIP checks the correct instantiation of the multiplicity and degree constraints of each connector motif. If errors exist, DesignBIP returns to the developer messages explaining the errors and pointers to the incorrect nodes of the model. In the case of a correct interaction model, DesignBIP returns an XML file with the generated code. In Appendix E, we present part of the generated XML code for the `Switchable Routes` example (Figure 9).

4.4 Model repositories

To promote reusability in DesignBIP, each project is accompanied by component type and coordination pattern [24] repositories. For instance, let us consider the mutual exclusion coordination pattern shown in Figure 11 that enforces the *no two processes can use the shared resource simultaneously* coordination property. The shared resource is managed by the unique —due to the cardinality being 1 —`Mutex Manager` component type. The multiplicities of all port types are 1 and therefore, all connectors are binary. The degree constraints require that each port instance of a component of type `Process` be attached to a single connector and each port instance of the `Mutex Manager` be attached to n connectors. The behaviors of the two component types enforce that once the resource is acquired by a component of type `Process`, it can only be released by the same component.

To use a coordination pattern, a developer needs to create an instance of the pattern in the model and evaluate its cardinality parameters. For instance, if the developer wants to enforce mutual exclusion on two instances of `Process` then n must be set equal to 2.

[3]The dash - indicates that p must not synchronize with any other port.

[4]The dash - indicates that p does not require any other port for synchronization.

Algorithm 1: Encoding a BIP Diagram into Require/Accept Macros

Data: Diagram $\mathscr{D} = \langle \mathscr{T}, \mathscr{C} \rangle$, where $\mathscr{C} = \{\Gamma_1, \ldots, \Gamma_n\}$ and $\Gamma = (a, \{m_p : d_p,\ t_p\}_{p \in a})$
Result: Returns the macros for each port type in $\mathscr{D}$
$require \longleftarrow \{\}$; $accept \longleftarrow \{\}$;
/* for each port type p in the diagram */
for $p \in \mathscr{T}.P$ **do**
  $require[p] = new\ Set\ ()$; $accept[p] = new\ Set\ ()$;
  /* for all connector motifs attached to p */
  for $\Gamma \in p.connectorMotifs$ **do**
    /* if the connector motif is singleton */
    if $|a| == 1$ **then**
      $require[p].add(-)$; $accept[p].add(-)$;
    else
      /* if the multiplicity of end attached to p is not 1, add all ports of the connector motif */
      if $m_p > 1$ **then**
        $accept[p].add(a)$;
      /* otherwise add all ports excluding p */
      else
        $accept[p].add(a \setminus \{p\})$;
      /* if the end attached to p is trigger */
      if $t_p ==$ trigger **then**
        $require[p].add(-)$;
      /* else if there exists at least one trigger */
      else if $\exists\ p \in a\ : t_p\ ==$ trigger **then**
        for $q \in a \wedge t_q ==$ trigger **do**
          /* for each trigger add an option */
          $require[p].add(p)$;
      /* else add all ports as many times as their multiplicity */
      else
        $optionRequire[p] = newList()$;
        for $q \in a \setminus \{p\}$ **do**
          $optionRequire[p].add(\underbrace{qq\ldots q}_{m_q})$;
        $optionRequire[p].add(\underbrace{pp\ldots p}_{m_p - 1})$;
        $require[p].add(optionRequire[p])$;
return *require* and *accept*;

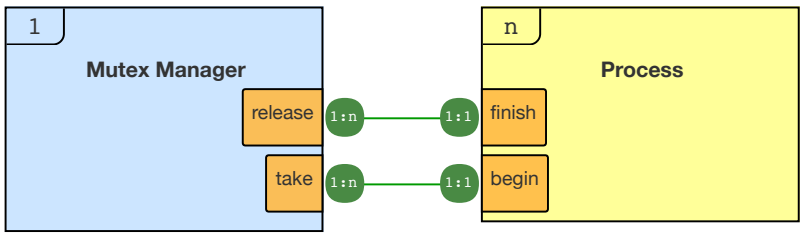


Figure 11: The mutual exclusion pattern

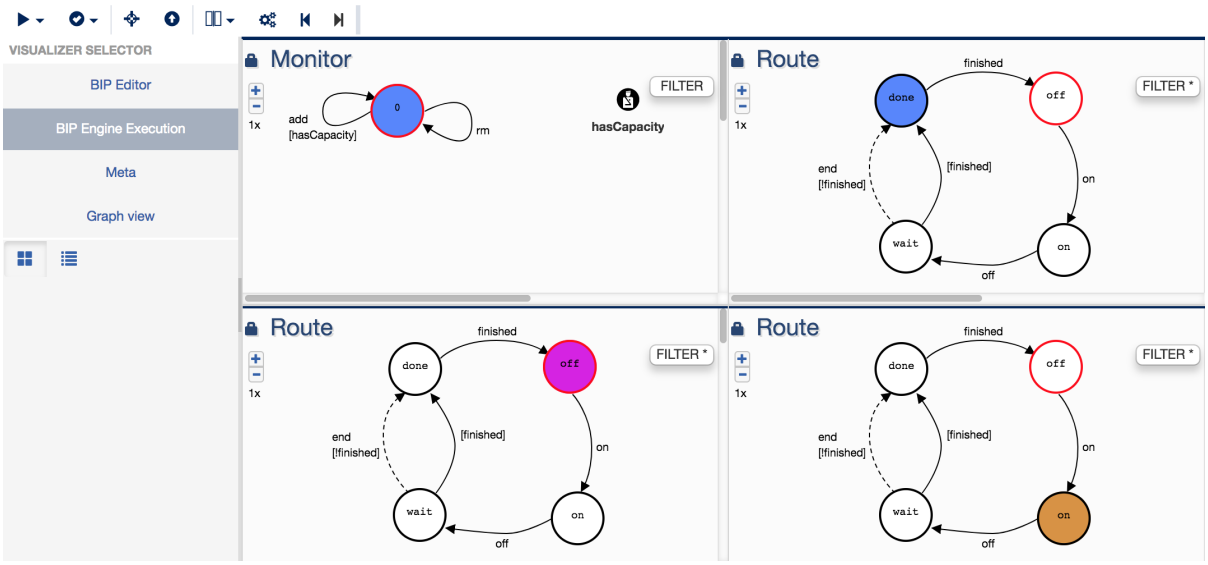


Figure 12: Visualization of the execution of the `Switchable Routes example`

5 JavaBIP-engine Execution and Visualization

After generating the system specification, the developer may use the integrated JavaBIP-engine to execute it. The JavaBIP-engine is offered through a dedicated plugin in DesignBIP. If the cardinality parameters of the component types have not been evaluated, the plugin asks the developer to provide the number of instances of each component type. It then instantiates the components and passes their reference to the JavaBIP-engine alongside with the generated Java and XML code. The plugin starts the JavaBIP-engine that runs the following three-step protocol in a cyclic manner: 1) upon reaching a state, each component notifies the JavaBIP-engine about possible outgoing transitions, 2) the JavaBIP-engine computes the possible interactions of the system, picks one, and notifies the involved components, 3) the notified components execute the functions associated with the corresponding transitions.

The output of the JavaBIP-engine (which transitions are picked at each execution cycle) is stored as a JSON object. When the plugin stops the execution of the engine (the execution time is defined by the developer), the output is sent back to the model editors of DesignBIP for simulation (see Figure 12). Initially, the developer picks the subset of components whose execution wants to simulate. Starting by highlighting the initial states of these components, the visualizer shows which transitions are executed in each execution cycle by firstly highlighting the fired transitions and finally their destination states.

6 Related Work

Model-driven component-based software engineering and development [6,12,17] has become an accepted practice for tackling software complexity in large-scale systems. It provides mechanisms to support design at the right level of abstraction, error detection, tool integration, verification and maintenance. Systems are built by composing and reusing small, tested building blocks called components.

The Generic Modeling Environment (GME) and its successor WebGME are open source Model Integrated Computing (MIC) tools developed for creating domain specific modeling environments and has been effectively applied to a number of domains [3,26,28,29].

Close to our approach is the ROSMOD design studio [20] that also relies on WebGME for collaborative code development and model editing features. The basic building blocks of ROSMOD are specified in its metamodel which is described in UML class diagrams [4]. The code development and compilation process have been integrated in the graphical user interface to keep the framework self-sufficient. ROSMOD integrates code development, code generation, compilation, run-time monitoring, and execution time plot generation. Nevertheless, in ROSMOD component behavior is defined directly with code and thus connection to verification tools is not supported.

A plethora of approaches exists for architecture specification. Patterns [14,18] are commonly used for specifying architectures in practical applications. The specification of architectures is usually done in a graphical way using general purpose graphical tools. Such specifications are easy to produce but their meaning may not be clear since the graphical conventions lack formal semantics and thus are not amenable to formal analysis. Significant work has been done by the Architecture Description Languages (ADLs) community. Many ADLs have been developed for architecture specification [25,27] with rigorous semantics that facilitate communication of system properties and allow system analysis. Nevertheless, according to [21], architectural languages used in practice mostly originate from industrial development (e.g., UML) instead of academic research (e.g., ADLs). Scientific questions remain about UML's formal properties [16]. The use of UML has been demonstrated in [13,19] for representing architectural concepts with a focus on the component and connector view. However, exploiting these constructors to express architecture views may result in a proliferation of models and stereotypes, which can be difficult to integrate into a well-structured code generation process. On the other hand, ADLs with formal semantics require the use of formal languages which are considered as challenging for practitioners to master [21]. We chose architecture diagrams, which rely on a small set of notions and combine the benefits of graphical languages and rigorous formal semantics.

7 Conclusion

We presented DesignBIP, which is a web-based, open source design studio[5] for modeling and generating BIP systems. To define system coordination aspects, we used a parameterized graphical language with formal semantics called architecture diagrams, which we extended with BIP coordination primitives. Designing and reusing models that are based on types and not on instances allowed us to cope with the issues of modeling complexity and size. We have implemented dedicated model/code editors, visualizers, as well as integrated the JavaBIP-engine. Additionally, we studied model transformations and implemented dedicated code generation plugins. We have opted for generating code from high-level graphical structures to avoid tedious and error-prone development of Boolean formulas. Rooting the whole modeling and execution process in rigorous semantics allows the connection to checkers and analysis tools. In the future, we are going to integrate data transfer information on connector motifs. We are also going to develop code generators for the BIP1 and BIP2 languages and integrate verification tools.

Acknowledgements

This research is supported by the National Science Foundation under award # CNS- 1521617. The authors would like to thank Hoang-Dung Tran and Venkataramana Nagarajan for helping with the implementation of DesignBIP.

[5] https://github.com/DesignBIP/DesignBIP

References

[1] *BIP Grammar*. http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/Bip2-simplified.html. Accessed: 2017-10-20.

[2] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen & Joseph Sifakis (2011): *Rigorous Component-Based System Design Using the BIP Framework*. Software, IEEE 28(3), pp. 41–48, doi:10.1109/MS.2011.27.

[3] Marco Beccani, Hakan Tunc, Addisu Taddese, Ekawahyu Susilo, Péter Völgyesi, Akos Lédeczi & Pietro Valdastri (2015): *Systematic design of medical capsule robots*. IEEE Design & Test 32(5), pp. 98–108.

[4] Donald Bell (2003): *UML basics: An introduction to the Unified Modeling Language*. The Rational Edge.

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

[6] Sami Beydeda, Matthias Book, Volker Gruhn et al. (2005): *Model-driven software development*. 15, Springer.

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

[8] Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek & Alina Zolotukhina (2017): *Exogenous coordination of concurrent software components with JavaBIP*. Software: Practice and Experience, pp. n/a–n/a, doi:10.1002/spe.2495. Available at http://dx.doi.org/10.1002/spe.2495. Spe.2495.

[9] Simon Bliudze & Joseph Sifakis (2007): *The Algebra of Connectors — Structuring Interaction in BIP*. In: Proc. of the EMSOFT'07, ACM SigBED, pp. 11–20, doi:10.1145/1289927.1289935.

[10] Simon Bliudze & Joseph Sifakis (2010): *Causal semantics for the algebra of connectors*. Formal methods in system design 36(2), pp. 167–194.

[11] Marius Bozga, Mohamad Jaber, Nikolaos Maris & Joseph Sifakis (2012): *Modeling Dynamic Architectures Using Dy-BIP*. In Thomas Gschwind, Flavio Paoli, Volker Gruhn & Matthias Book, editors: Software Composition, Lecture Notes in Computer Science 7306, Springer Berlin Heidelberg, pp. 1–16, doi:10.1007/978-3-642-30564-1_1. Available at http://dx.doi.org/10.1007/978-3-642-30564-1_1.

[12] Szyperski Clemens, Gruntz Dominik & Murer Stephan (1998): *Component software: Beyond object-oriented programming*. Addison-Wesley.

[13] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers & Reed Little (2002): *Documenting software architectures: views and beyond*. Pearson Education.

[14] Robert Daigneau (2011): *Service design patterns: Fundamental design solutions for SOAP/WSDL and restful Web Services*. Addison-Wesley.

[15] Romain Edelmann, Simon Bliudze & Joseph Sifakis (2016): *Functional BIP: Embedding Connectors in Functional Programming Languages*. In: 17^{th} Symposium on Trends in Functional Programming.

[16] David Harel & Bernhard Rumpe (2004): *Meaningful modeling: what's the semantics of "semantics"?* Computer 37(10), pp. 64–72.

[17] George T Heineman & William T Councill (2001): *Component-based software engineering*. Springer.

[18] Gregor Hohpe & Bobby Woolf (2003): *Enterprise integration patterns: designing, building, and deploying messaging solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[19] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl & Jaime R Silva (2004): *Documenting component and connector views with UML 2.0*. Technical Report, DTIC Document.

[20] Pranav Srinivas Kumar, William Emfinger, Amogh Kulkarni, Gabor Karsai, Dexter Watkins, Benjamin Gasser, Cameron Ridgewell & Amrutur Anilkumar (2015): *ROSMOD: A Toolsuite for Modeling, Generating, Deploying, and Managing Distributed Real-time Component-based Software using ROS*. In: Rapid System Prototyping Symposium (ESWEEK), Amsterdam, The Netherlands.

[21] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione & Anthony Tang (2013): *What industry needs from architectural languages: A survey*. Software Engineering, IEEE Transactions on 39(6), pp. 869–891.

[22] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky & Ákos Lédeczi (2014): *Next Generation (Meta) Modeling: Web-and Cloud-based Collaborative Tool Infrastructure.* In: *MPM@ MoDELS*, pp. 41–60.

[23] Anastasia Mavridou, Eduard Baranov, Simon Bliudze & Joseph Sifakis (2016): *Architecture Diagrams: A Graphical Language for Architecture Style Specification*. In: Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016., pp. 83–97, doi:10.4204/EPTCS.223.6. Available at `http://dx.doi.org/10.4204/EPTCS.223.6`.

[24] Anastasia Mavridou, Stachtiari Emmanouela, Simon Bliudze, Anton Ivanov, Panagiotis Katsaros & Joseph Sifakis (2016): *Architecture-based Design: A Satellite On-board Software Case Study*. In: Proceedings of the 13th International Conference on Formal Aspects of Component Software (FACS), pp. 260–279.

[25] Nenad Medvidovic & Richard N Taylor (2000): *A classification and comparison framework for software architecture description languages*. IEEE transactions on Software Engineering 26(1), pp. 70–93.

[26] Himanshu Neema, Janos Sztipanovits, Martin Burns & Edward Griffor (2016): *C2WT-TE: A model-based open platform for integrated simulations of transactive smart grids*. In: Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2016 Workshop on, IEEE, pp. 1–6.

[27] Mert Ozkaya & Christos Kloukinas (2013): *Are we there yet? Analyzing architecture description languages for formal analysis, usability, and realizability*. In: Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on, IEEE, pp. 177–184, doi:10.1109/SEAA.2013.34.

[28] John A Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey & Brian Ellis (2003): *Vest: An aspect-based composition tool for real-time systems*. In: Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE, IEEE, pp. 58–69.

[29] Kleanthis Thramboulidis (2005): *Model-integrated mechatronics-toward a new paradigm in the development of manufacturing systems*. IEEE Transactions on Industrial Informatics 1(1), pp. 54–61.

A First-order Interaction Logic (FOIL) and Require/Accept Macro-notation

We assume that a finite set of component types $\mathscr{T} = \{T_1, \ldots, T_n\}$ is given. Instances of a component type T have the same interface and behavior. We write $c:T$ to denote that a component c is of type T. We denote by $T.p$ the *port type* p, i.e., a port belonging to the interface of type T. We write $\mathscr{T}.P$ 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. The first-order interaction logic (FOIL) is defined by the grammar:

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

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 $Pr = true$). We define the usual notation for the universal quantifier:

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

The semantics of FOIL 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 $\mathscr{T} = \{T_1, \ldots, T_n\}$ is given. An architecture is a pair $\langle \mathscr{B}, \gamma \rangle$, where $\mathscr{B}$ is a set of component instances of types from $\mathscr{T}$ and γ is a set of interactions 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:

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

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

Example A.1. *The* Star architecture style*, for any number of components of type S, can be expressed in FOIL as follows:*

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

Two macros are used: 1) the **Require** macro and 2) the **Accept** macro to define synchronization constraints in JavaBIP.

Require defines ports required for interaction. Let $T_1, T_2 \in \mathscr{T}$ be two component types. For instance:

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

means that, to participate in an interaction, each of the ports p of component instances of type T_1 requires either the participation of *precisely two* of the ports q of component instances of type T_2 or one instance of r. Notice the semicolon in the macro that separates the two options.

Accept defines optional ports for participation, i.e., it defines the boundary of interactions. This is expressed by explicitly excluding from interactions all the ports that are not optional. Let $T_1, T_2 \in \mathscr{T}$. For instance:

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

B The DesignBIP Metamodel

Figure 13 shows part of the DesignBIP metamodel as a UML class diagram. For ease of presentation, we omitted from Figure 13 composite component types and exported ports. A `BIP_project` contains component types, connector motifs , as well as formalized design patterns, i.e., *architecture styles* and component type repositories. A `BIP_project` has three attributes: 1) a `name`, 2) a functionality `description` and 3) the `engineOutput` in the form of a JSON object that can be used for visualization purposes.

A `BIP_project` contains a set of `Component_Type` elements that have three attributes: 1) `name`, 2) a `cardinality` parameter that denotes the number of component instances of this type and 3) a `definition` of variables. The behavior of a `Component_Type` is described, as explained in Section 2, by an LTS with three types of transitions: `Internal_Transition`, `Spontaneous_Transition`

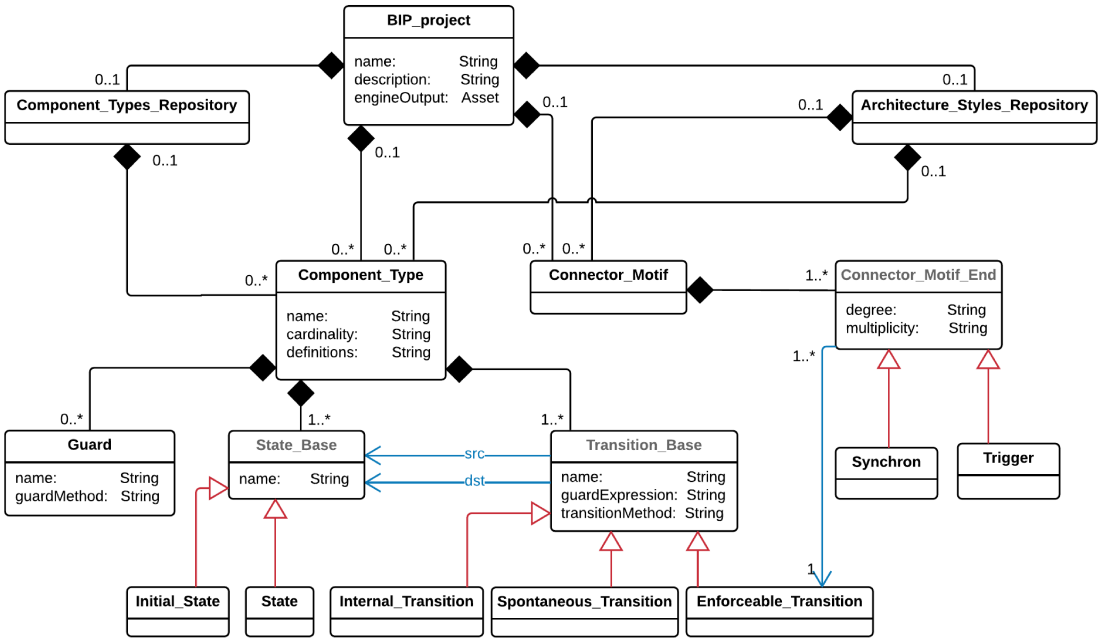


Figure 13: Part of the DesignBIP metamodel

and `Enforceable_Transition`. Enforceable transitions are exported as ports. The `State_Base` and `Transition_Base` elements are abstract. A transition is characterized by three attributes: 1) `name`, 2) a `guardExpression` and 3) a `transitionMethod` that describes the action performed upon the execution of the transition. The `guardExpression` is a Boolean expression involving the conjunction '&', disjunction '|' and negation '!' operators, parenthesis and the names of `Guard` elements. A `Guard` is evaluated through its `guardMethod` attribute.

Additionally, `BIP_project` contains a set of `Connector_Motif` elements that have two attributes 1) `multiplicity` and 2) `degree`. A `Connector_Motif` contains a non-empty set of `Connector_Motif_End` elements which can be of two types: `Synchron` or `Trigger`. Each `Connector_Motif_End` is associated with a single component port, i.e., an `Enforceable_Transition`. More than one `Connector_Motif_End` can be associated with the same `Enforceable_Transition`.

C Proof of Proposition 3.1

Proof of Proposition 3.1. Necessity $\rightarrow$: Assume that there exists a single architecture $\langle \mathscr{B}, \gamma \rangle$ conforming to the diagram. Condition 1 is trivially obtained: $n_p < m_p$ cannot occur as the multiplicity of ports cannot be greater than the number of component instances. Notice that $\prod_{q \in a} \binom{n_q}{m_q}$ is equal to the number of different ways one could connect instances of port types in a, so that port $p \in a$ has n_p instances and connector k contains $m_{k,p}$ ports $p_i \in p$. Assume that the conforming architecture has s connectors. Then $\forall p \in a$, $s_p = s$, otherwise there would exist zero connectors. Also, $s \leq \prod_{q \in a} \binom{n_q}{m_q}$ otherwise, by the pigeonhole principle there would exist duplicated connectors. Next, we show that if $s < \prod_{q \in a} \binom{n_q}{m_q}$, then the simple architecture diagram has multiple conforming architectures. From $s < \prod_{q \in a} \binom{n_q}{m_q}$, it follows that there exists a connector $k^* \notin \gamma$. We are going to show how to construct another conforming

architecture that contains k^*. Let $k' \in \gamma$ be an arbitrary connector. For each port type p such that $n_p > 1$ and k' and k^* do not contain the same set of instances of p, we take the following steps. Consider instances $p_i \in k' \setminus k^*$ and $p_j \in k^* \setminus k'$ of port type p. From Lemma C.1, it follows that there exists a connector $k \in \gamma$ that contains p_j but not p_i. Then, we can transform the architecture by letting k contain p_i instead of p_j, and letting k' contain p_j instead of p_i. Clearly, the architecture still conforms to the diagram after this transformation. If k' and k^* still do not contain the same set of instances of p, then repeat the previous step with another pair of instances $p_i \in k' \setminus k^*$ and $p_j \in k^* \setminus k'$ of type p. If they contain the same set of instances, then continue with another port type. After we have finished with all port types, the resulting architecture still conforms to the diagram but now contains $k' = k^*$, which implies that it cannot be identical to the original architecture. Therefore, we have shown that if $s < \prod_{q \in a} \binom{n_q}{m_q}$, then a simple architecture diagram has multiple conforming architectures, which concludes the proof of necessity.

Sufficiency $\leftarrow$: We show that for each connector motif, there exists a unique configuration conforming to the architecture diagram. Since each connector must contain m_q instances of port type q out of the existing n_q instances, the number of possible connectors for a given motif is $\prod_{q \in a} \binom{n_q}{m_q}$. Notice that since $m_p \leq n_p$ for every port type p, the set of possible connectors is non-empty. A conforming configuration must have this many connectors according to our assumption $s_p = \prod_{q \in a} \binom{n_q}{m_q}$, which implies that a conforming configuration must contain all possible connectors. Since this set is unique for each connector motif, the architecture itself must also be unique. □

Lemma C.1. *Let $k \in \gamma$ be a connector of an architecture $\langle \mathscr{B}, \gamma \rangle$ conforming to a simple architecture diagram that contains instance p_i but not instance p_j of port type p. Then, there exists another connector $k' \in \gamma$ that contains instance p_j but not instance p_i of port type p.*

Proof. For the sake of contradiction, suppose that every connector that contains p_j also contains p_i. Since p_i is also contained by connector k, this would imply that p_i is contained by more connectors than p_j, which contradicts the semantics of simple architecture diagrams. Hence, the claim holds. □

D Generated Java Code for the Route Component Type

```
@Ports({
   @Port(name = "end", type = PortType.spontaneous),
   @Port(name = "on",  type = PortType.enforceable),
   @Port(name = "off", type = PortType.enforceable),
   @Port(name = "finished", type = PortType.enforceable)
})
@ComponentType(initial = "off", name = "Route")
public class Route implements CamelContextAware {
   private CamelContext camelContext;
   private String routeId;
   private int deltaMemory = 100;

   public Route(String routeId, CamelContext camelContext) {
     this.routeId = routeId;
     this.camelContext = camelContext;
   }

   @Transition(name = "on", source = "off", target = "on")
   public void startRoute() throws Exception {
     camelContext.resumeRoute(routeId);
   }
```

```
  @Transition(name = "off", source = "on", target = "wait")
  public void stopRoute() throws Exception {
    camelContext.suspendRoute(routeId);
  }

  @Transition(name = "end", source = "wait",
  target = "done", guard = "!finished")
  public void spontaneousEnd() {}

  @Transition(name = "", source = "wait",
  target = "done", guard = "finished")
  public void internalEnd() {}

  @Transition(name = "finished", source = "done",
  target = "off")
  public void finishedTransition() {}

  @Guard(name = "finished")
  public boolean isFinished() {
    return camelContext.getInflightRepository().
      size(camelContext.getRoute(routeId).
      getEndpoint()) == 0;
  }
}
```

E Part of the Generated XML code for the Switchable Routes Example

```
1  <require>
2    <effect id="on" specType="Route"/>
3    <causes>
4       <port id="add" specType="Monitor"/>
5    </causes>
6  </require>
7  <accept>
8    <effect id="on" specType="Route"/>
9    <causes>
10     <port id="add" specType="Monitor"/>
11   </causes>
12 </accept>
13 <require>
14    <effect id="add" specType="Monitor"/>
15    <causes>
16       <port id="on" specType="Route"/>
18    </causes>
19 </require>
20 <accept>
21   <effect id="add" specType="Monitor"/>
22   <causes>
23       <port id="on" specType="Route"/>
24   </causes>
25 </accept>
26 <require>
27    <effect id="off" specType="Route"/>
28    <causes>
29    </causes>
30 </require>
31 <accept>
32    <effect id="off" specType="Route"/>
33    <causes>
34    </causes>
35 </accept>
```

F Demonstration of DesignBIP

We provide a step-by-step demonstration of the DesignBIP tool. In particular, we demonstrate how to model behavior and interaction, generate the Java and XML code and execute the system using the JavaBIP-engine.

F.1 Step 1: Presentation of the DesignBIP Environment

DesignBIP is a web-based, open-source tool. The initial page of DesignBIP is shown in Figure 14. The developer may navigate to the Projects page by double-clicking on the `Projects` icon.

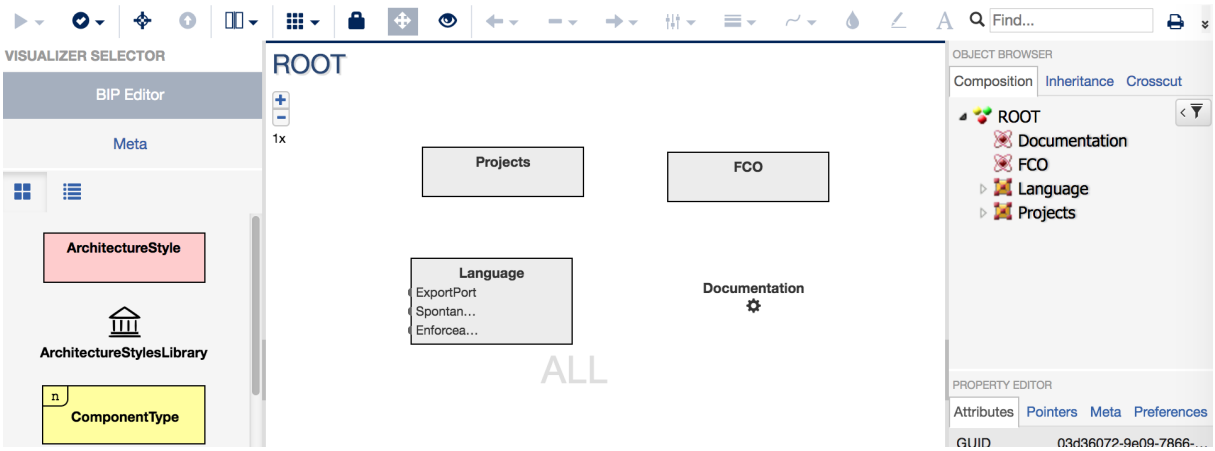


Figure 14: Initial page of DesignBIP

The `Projects` page is shown in Figure 15. The center panel is the Canvas. The left side of the screen shows the `Visualizer Selector` options and below them is the `Part Browser`, which displays the concepts than can be instantiated inside the Canvas. For instance, dragging and dropping a `Project` from the `Part Browser` creates a new project on the Canvas. The top right corner of the user interface is the `Object Browser`, which shows the composition hierarchy of DesignBIP starting at Root. Embeddable documentation can be added at every level of this hierarchy. Below the `Object Browser` is the `Property Editor`, where attributes, preferences, and other properties of the currently selected project can be edited.

DesignBIP provides a collaborative and versioned environment. Multiple developers can collaborate on the same smart contract simultaneously. Changes are immediately broadcasted to all developers and everyone sees the same state. This is similar to how Google Docs works, except that the models in DesignBIP have a much richer data model, which makes consistency management more challenging. Changes in DesignBIP are committed and versioned, which enables branching, merging, and viewing the history of a contract. Figure 16 shows the history of the `master` branch. During the demonstration, we are going to show how a developer can create a new branch and merge it into the master branch.

F.2 Step 2: Presentation of the DesignBIP Language

Next, we present some elements of the BIP language, which is defined in DesignBIP as a UML class diagram (Figure 17). By double-clicking `Meta` in the `Visualizer Selector` shown in Figure 15, we navigate to the language-specification of DesignBIP. As shown in Figure 17, the behavior of a

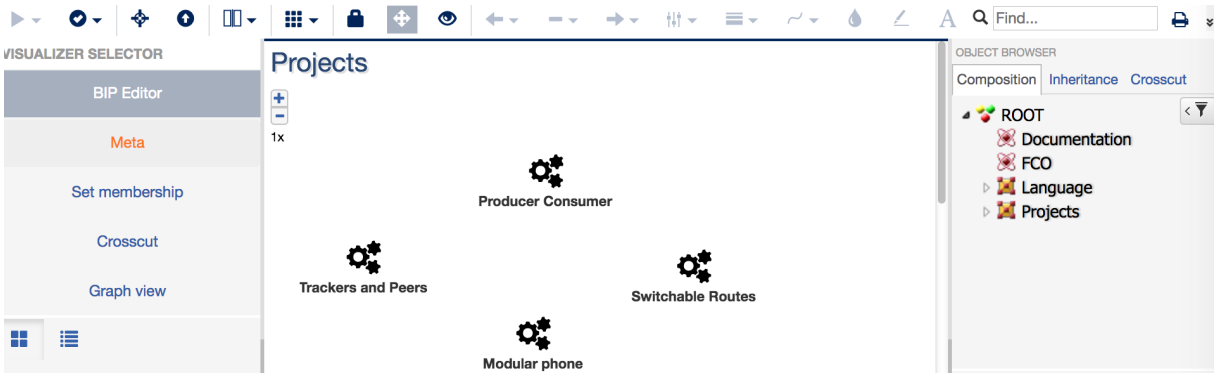


Figure 15: The `Projects` page

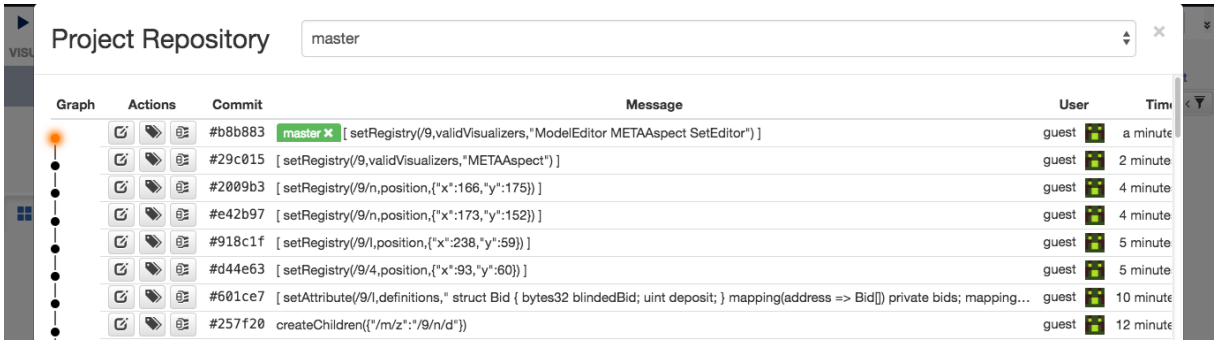


Figure 16: Versioning in DesignBIP

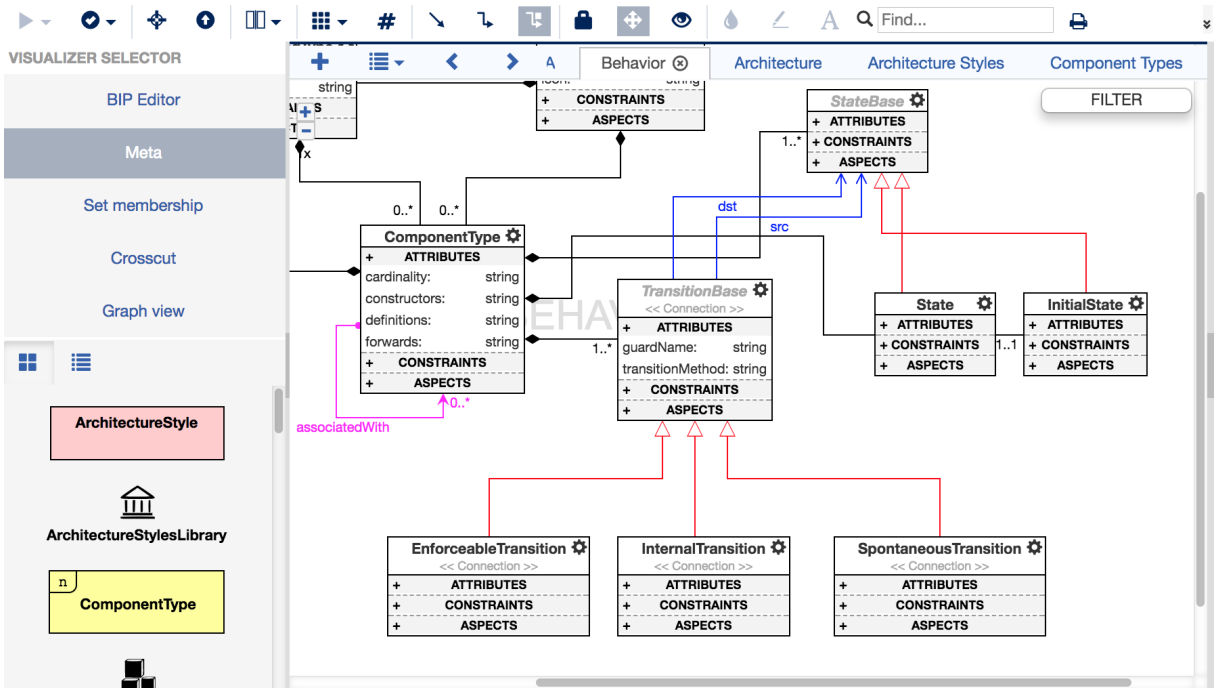


Figure 17: The DesignBIP language

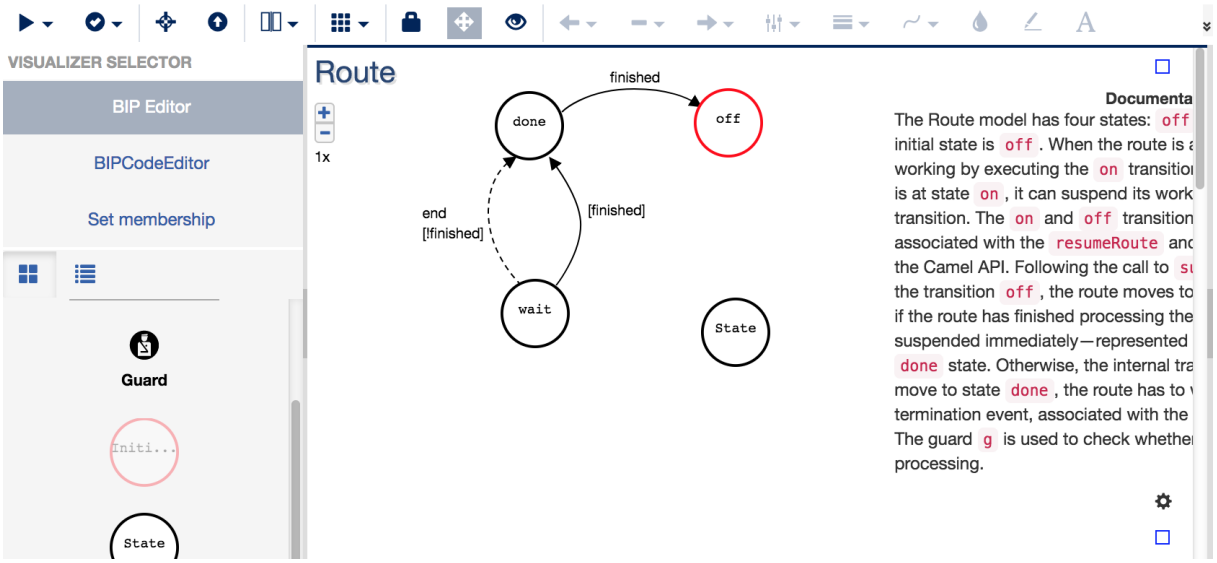


Figure 18: Designing an LTS in DesignBIP

`ComponentType` is described as an LTS. The `State_Base` element is abstract and it can be instantiated by either an `InitialState` or a `State`. Each contract must have exactly one `InitialState`, which is enforced by the cardinality of the containment relation. Similarly, the `Transition_Base` element is abstract and it can be instantiated by either an `EnforceableTransition`, an `InternalTransition`, or an `SpontaneousTransition`. The developer can see other aspects of the DesignBIP metamodel by clicking the `Architecture`, `Architecture Styles` and `Component Types` tabs.

F.3 Step 3: Designing a BIP System

Next, we present how to design a BIP system using DesignBIP. After creating a new `Project` on the canvas shown in Figure 15 (as described in Step 1), the developer must specify a unique name for this project, e.g., `SwitchableRoutes`, in the `name` attribute of the Project listed in the `Property Editor`. Then, the developer creates the states of the LTS (Figure 18) by drag-and-dropping states from the `Part Browser`. By clicking on a state, the developer may add a transition to another state or to the same state. For each transition, the developer may specify its corresponding attributes, i.e., `guardName` and `transitionMethod` at the `Property Editor` or at the dedicated Java code editor that we have developed. The developer may navigate to the code editor by double-clicking `BIPCodeEditor` in the `Visualizer Selector`.

Figures 18,19, and 20 show the model/code editors and the representation of the switchable routes project. From the LTS shown in the LTS model editor (Figure 18), we generate equivalent Java code, which consists in the darker parts of the code shown in the code editor (Figure 19). This part of the code is generated automatically from the LTS and cannot be altered by the developer in the code editor. The developer can only change the LTS in the model editor. The lighter parts of the code in the code editor represent parts of the code that the developer has directly defined in the code editor. For instance, the developer may specify variables, transition statements and guards, directly in the code editor. The code and model editors are tightly integrated. Once a developer makes a change in the model editor, the code editor is updated and vice versa. Figure 20 shows the dedicated model editor for BIP architecture

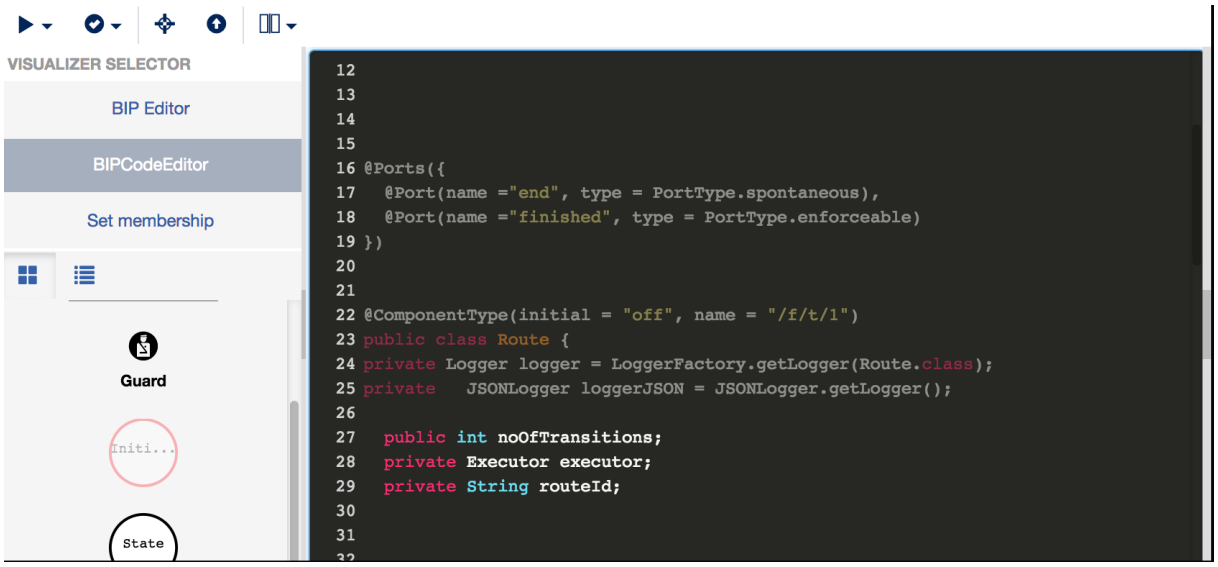


Figure 19: The DesignBIP code editor

diagrams. As mentioned earlier, documentation can be added at every level of DesignBIP.

F.4 Step 4: Generating Java and XML Code

To generate Java and XML code a developer must click on the `BehaviorCodeGenerator` and `ArchitectureCodeGenerator` plugins offered by the drop-down menu in the upper left corner of the tool (Figure 21). Then, the widget shown in Figure 22 pops up, and the developer may click on the `Save & Run` button to continue with the code generation.

If the code generation is successful, i.e., there are no specification errors in the given input, then the widget shown in Figure 23 pops-up. The developer may then click on the generated artifacts to download the generated Java or XML code. If, on the other hand, the code generation is not successful due to incorrect input, then, the widget shown in Figure 24 pops-up. As shown in Figure 24, DesignBIP lists the errors found in the specification of the contract with detailed explanatory messages. Additionally, DesignBIP provides links (through `Show node`) that redirect the developer to the erroneous nodes of the contract.

F.5 Step 5: Running the Integrated JavaBIP-engine

If the behavior and interaction code generation has finished successfully, the developer may execute the generated BIP system using the integrated JavaBIP-engine. To do that, the developer must click on the `JavaBIPEngine` plugin in the drop-down menu shown in Figure 21. Then, the developer must instantiate any parameters of the model in the widget shown in Figure 25 and also must specify how many transitions should be executed by the JavaBIP-engine. Then, the developer may click on the `Save & Run` button to continue with the execution. Once the JavaBIP-engine has finished its execution, the developer may visualize the output of the engine by clicking `BIP Engine Execution` in the `Visualizer Selector` panel (see Figure 20). Then, the widget shown in Figure 26 appears, where the developer specifies the number of instances of each component type to be visualized in the simulated output. Finally, the visualization of the output is as described in Section 5.

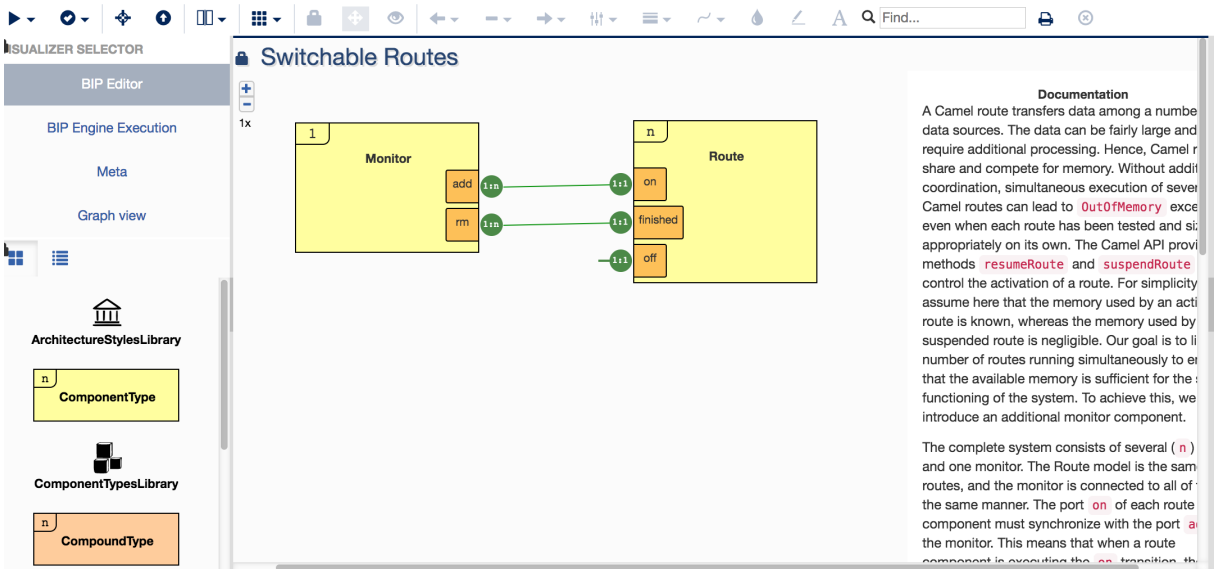


Figure 20: The BIP model editor

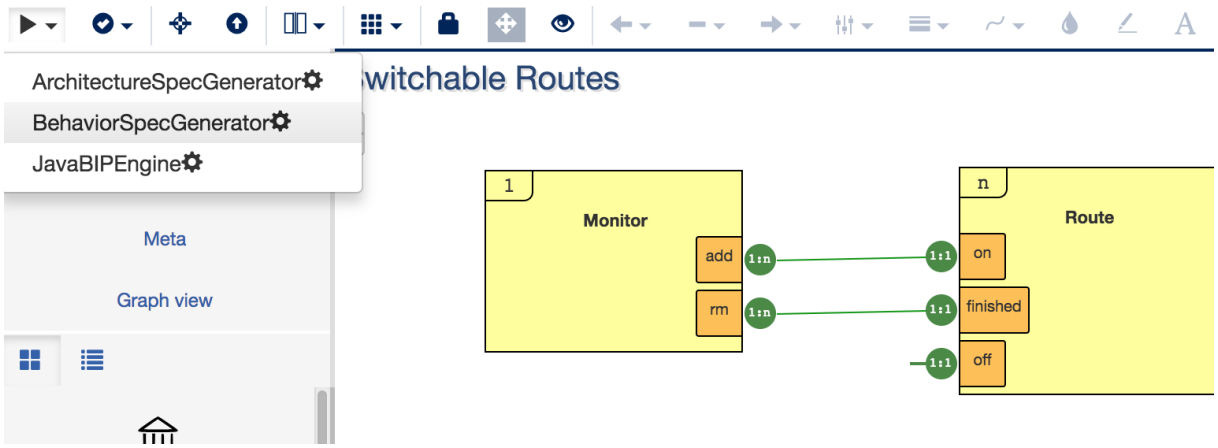


Figure 21: DesignBIP plugins

Figure 22: Widget for code generation plugins

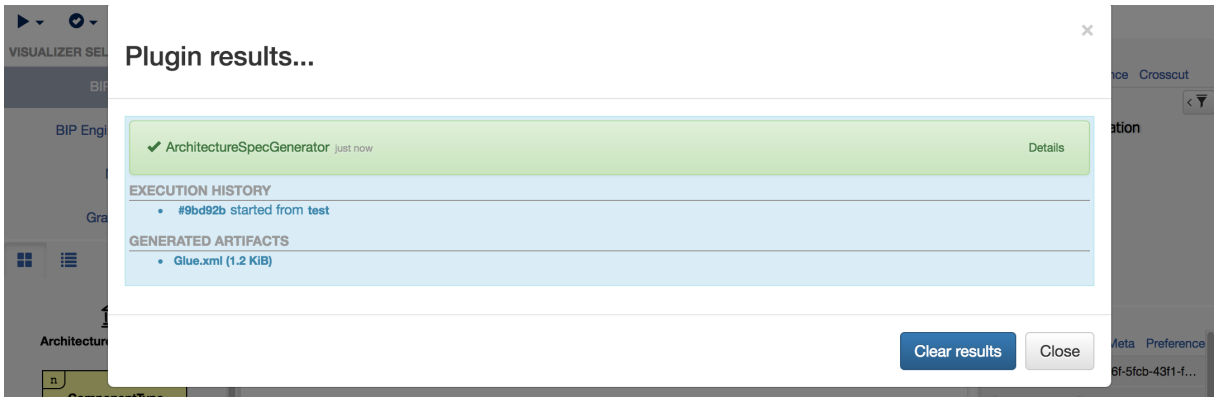


Figure 23: Successful code generation

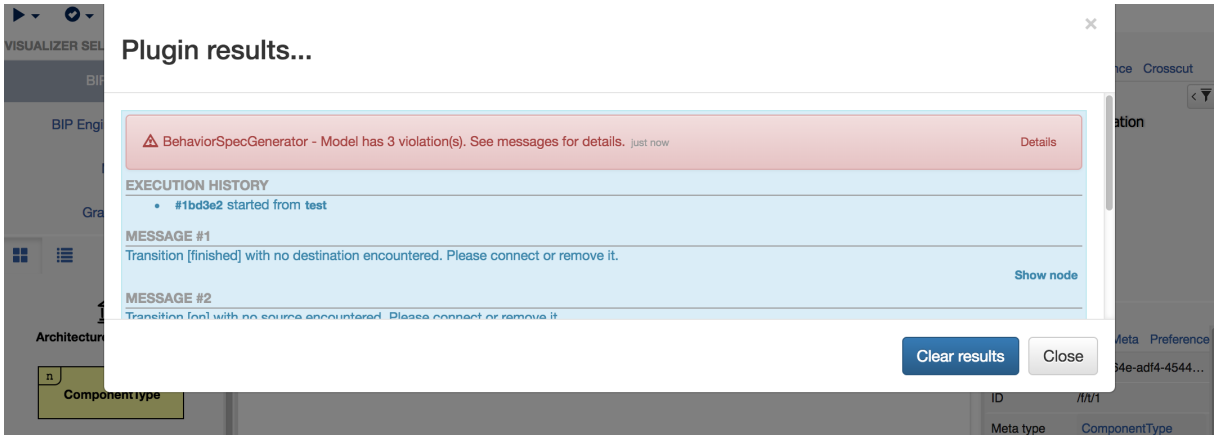


Figure 24: Unsuccessful code generation due to incorrect input

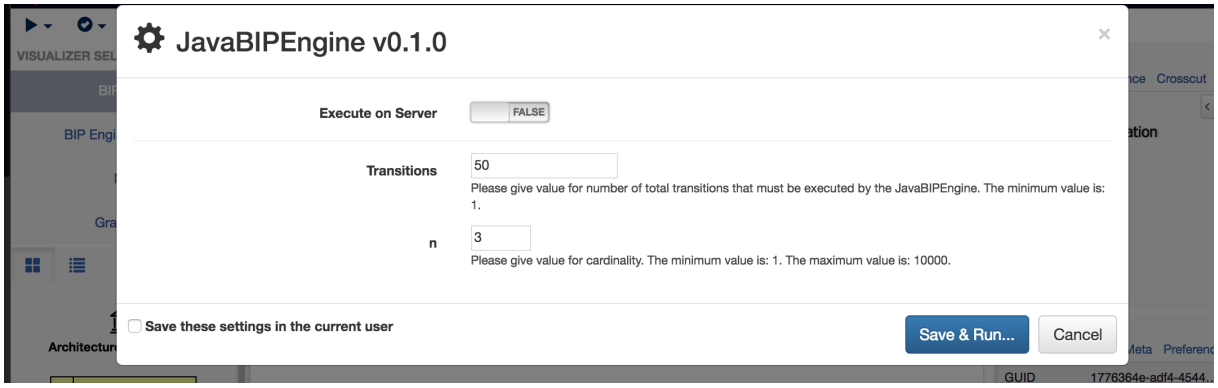


Figure 25: Widget for JavaBIP-engine plugin

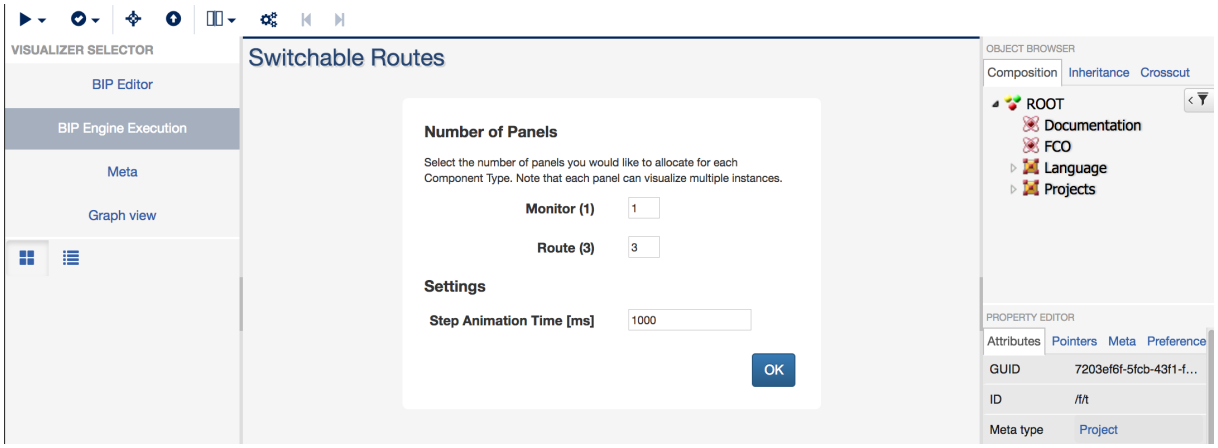


Figure 26: Visualization of the JavaBIP-engine output