Modularizing Behavioral and Architectural Crosscutting Concerns in Formal Component-Based Systems - Application to the Behavior Interaction Priority Framework

Antoine El-Hokayem, Yliès Falcone, Mohamad Jaber

Modularizing Behavioral and Architectural Crosscutting Concerns in Formal Component-Based Systems

Application to the Behavior Interaction Priority Framework

Antoine El-Hokayem[a], Yliès Falcone[a,*], Mohamad Jaber[b]

[a] *Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France*
[b] *American University of Beirut, Beirut, Lebanon*

Abstract

We define a method to modularize crosscutting concerns in Component-Based Systems (CBSs) expressed using the Behavior Interaction Priority (BIP) framework. Our method is inspired from the Aspect Oriented Programming (AOP) paradigm which was initially conceived to support the separation of concerns during the development of monolithic systems. BIP has a formal operational semantics and makes a clear separation between architecture and behavior to allow for compositional and incremental design and analysis of systems. We distinguish local from global aspects. Local aspects model concerns at the component level and are used to refine the behavior of components. Global aspects model concerns at the architecture level, and hence refine communications (synchronization and data transfer) between components. We formalize local and global aspects as well as their composition and integration into a BIP system through rigorous transformation primitives. We present `AOP-BIP`, a tool for Aspect-Oriented Programming of BIP systems, demonstrate its use to modularize logging, security, and fault tolerance in a network protocol, and discuss its possible use in runtime verification of CBSs.

1. Introduction

A component-based approach [5, 38, 50] consists in building complex systems by composing components (building blocks). Such approach confers numerous advantages (e.g., productivity, incremental construction, compositionality) that facilitate dealing with the complexity in the construction phase. Component-based design is based on the separation between interaction and computation. The isolation of interaction mechanisms allows for a global treatment and analysis on interactions between components even if local computations on components are not visible (i.e., when components are "black boxes").

A typical system consists of its main logic along with tangled code that implements multiple other functionalities. Such functionalities are often seen as secondary to the system. For example, logging is not particularly related to the main logic of most systems, yet it is often scattered throughout multiple locations in the code. Logging and the main code are separate domains and

*Corresponding author

Email addresses: `antoine.el-hokayem@univ-grenoble-alpes.fr` (Antoine El-Hokayem), `ylies.falcone@univ-grenoble-alpes.fr` (Yliès Falcone), `mj54@aub.edu.lb` (Mohamad Jaber)

represent different *concerns*. A concern is defined in [15] as a "*domain used as a decomposition criterion for a system or another domain with that concern*". Domains include logging, persistence, and system policies such as security. Concerns are often found in different parts of a system, or in some cases multiple concerns overlap one region. These are called *crosscutting concerns*. Aspect-Oriented Programming (AOP) [34, 52] aims at *modularizing* crosscutting concerns by identifying a clear role for each of them in the system, implementing each concern in a separate module, and loosely coupling each module to only a limited number of other modules. Essentially, AOP defines mechanisms to determine the locations of the concerns in the system execution by introducing *joinpoints* and *pointcuts*. Then, it determines what to do at these locations by introducing *advices*. Finally, it provides a mechanism to coordinate all the advices happening at a location by introducing a process called *weaving*.

Motivations and challenges. In Component Based Systems (CBSs), crosscutting concerns arise at the levels of components [19, 37] (building blocks) and architectures (communications). Integrating crosscutting concerns in CBSs allows users to reason about crosscutting concerns in separation, and favors correct-by-construction design. Defining an AOP paradigm for CBSs poses multiple challenges. While the execution of a sequential program can be seen as a sequence of instructions, the semantics of a CBS is generally more complex and relies on a notion of architecture imposing several constraints on their execution. As a consequence of these constraints, AOP matching, instrumentation and modifications need to be extended to account for the architecture (i.e., data transfer and rendez-vous between components). Multiple approaches [44, 11, 19, 37, 16] have sought to apply AOP for CBSs. However they have not formalized the AOP concepts in the context of CBSs, nor have they defined formal transformations and semantics that allow us to reason about the transformed systems rigorously.

Approach overview. We formalize AOP for CBSs. We rely on a general abstraction of CBS executions as traces, AOP is then concerned with matching segments of the trace and modifying them. We identify two views for CBSs: local and global. The local view is concerned with the behavior of a component, the component is seen as a white box and its internals are inspected. The global view is concerned with the architecture of the system, i.e., the interaction between components and their interfaces, the components are seen as black boxes. We formalize our approach by extending an existing formalism of CBSs: the Behavior Interaction Priority (BIP) framework [5, 42]. The BIP component-based framework uses formal operational semantics. BIP consists of: (1) Behavior which is handled by atomic components; (2) Interaction which describes the collaboration between the atomic components; (3) Priority determines which interaction to execute out of many. Multiple formalizations for CBS exist such as BRIC [17], Pi-Calculus [39], and Fractal [11]. The choice of BIP is mainly due to compatibility. BIP models explicitly the two views [48], behavior models the local view while interaction and priority model the global architectural view. Furthermore, BIP can also be viewed as an architecture description language (ADL) [38], and is used for systems modeling [5]. In particular, an AADL model (which is a superset of ADL [27]) can be transformed into a BIP model [13]. In addition, BIP supports a full set of tools [51] for manipulating the BIP model, source-to-source transformations, model transformations, code generation, and compositional verification [7]. We implement our approach as a model-to-model transformation tool. We transform an existing BIP model using an AOP description to a new BIP model that implements it. For each view, we define pointcuts to target transitions and interactions as joinpoints, and modify them by appending additional computation before and after their execution. Additionally, we allow advices to change

the state of atomic components. As such, we are able to implement logging, authentication, congestion control and fault tolerance to a simple network protocol (Section 8.2). In Section 8.3, we show the application to runtime verification [25, 35]. Our approach can also be used for various crosscutting concerns in CBSs such as testing, runtime enforcement [24] and monitor synthesis [12]. However, we note that our approach is intended to cover a basic set of advices and not all. For example, we do not allow for a change in priorities (and thus the scheduling) between the interactions of components. We also do not allow for the disabling of interactions. These advices can be implemented by defining more transformations following our methodology.

Paper Structure. We begin by presenting the concepts of the BIP framework and AOP in Section 2 and Section 3, respectively. In Section 4, we formalize the identification and description of concerns in the context of BIP. In general, concerns are expressed by determining their locations in the system, and their behavior at the given locations. Based on the formalization of concerns, we determine the *rules* that govern the integration of these concerns in a BIP system. Therefore, given an initial BIP system, and a description of concerns, we transform it so as to include the desired concerns. We distinguish and define two types of aspects: *Global* and *Local*, in Sections 5 and 6, respectively. Global aspects are used to model crosscutting concerns at the architecture level and are thus useful to refine communications (synchronization and data transfer) between components. Local aspects are used to model crosscutting concerns within components and are thus particularly useful to refine the behaviors of components. Moreover, in Section 7, we define the notion of aspect container which serves as a construct for grouping aspects. We discuss weaving strategies of aspects and their integration into a BIP system. Moreover, we define a high-level language for writing local, global aspects, and aspect containers. Our framework is fully implemented in `AOP-BIP` and tested on a network protocol refined to add several crosscutting concerns: logging, authentication, congestion control and fail-safe (Section 8). Furthermore, we discuss monitoring CBSs with our approach, since runtime verification is a crosscutting concern. Finally, we present related work in Section 9, then draw conclusions and present future work in Section 10.

This paper revises and significantly extends a paper that appeared in the proceedings of the 14th international conference on Software Engineering and Formal Methods (SEFM 2016) [20]. The additional contributions can be summarized as follows. First, we elaborate the general overview of our approach, and explain the applicability of our approach to other formalizations of CBSs (Section 4). Second, we present the full description of local aspects (Section 6) designed to target the local view. Third, we define the strategies for weaving aspects (Section 7), by introducing containers and weaving procedures. Fourth, we extend the experimental work to include a case study on using our approach for runtime verification of CBSs (Section 8.3). Finally, we improve the presentation and readability by adding more details and examples, elaborating on the views (Section 4.2), and improving on the notation.

2. Behavior Interaction Priority (BIP)

Behavior Interaction Priority (BIP) [5, 42] allows to define systems as sets of atomic components with prioritized interactions. We present components, interactions, priorities, and their composition. First, we provide definitions for the construction of BIP systems, and then illustrate them with Example 1.

We begin by describing the *update function*. An update function abstracts code execution, which may modify the state of the system by reading and writing to variables.

Definition 1 (Update function). *An update function F over a set of variables X is a sequence of assignments $x_1 := expr_{X_1} \cdots x_n := expr_{X_n}$ such that $\forall i \in [1,n] : x_i \in X$, and $expr_{X_i}$ is an expression using variables in set X_i, for $i \in [1,n]$.*

The set of all sequences of update functions is denoted by $\mathcal{F}$. Furthermore, for $F = x_1 := expr_{X_1} \cdots x_n := expr_{X_n}$, we use $\mathrm{readvar}(F)$ and $\mathrm{writevar}(F)$ to denote variable the *read* and *modified* variables, i.e., $\cup_{i\in[1,n]} X_i$ and $\cup_{i\in[1,n]}\{x_i\}$, respectively. Such sets can be obtained using a simple and standard syntactic analysis of the update function. Moreover, for two update functions F_1 and F_2, we note $F_1 \cdot F_2$, update function formed by the concatenation of F_1 and F_2 (noted $F_1\ F_2$ to lighten the notation at places).

An atomic component is the basic computation unit. It is defined by its interface (i.e., a set of ports) and behavior defined as a Labeled Transition System (LTS) extended with data. Transitions are labeled with update functions, guards, and ports. Ports define communication and synchronization points for components. A port can be associated with some variables (of the component), to exchange data with other components. Ports are said to be exported by the component as they define its interface.

Definition 2 (Port). *A port $\langle p, X_p\rangle$ is defined by an identifier p and a set of attached local variables X_p (denoted by $p.vars$).*

Definition 3 (Atomic component). *An atomic component is a tuple $\langle P, L, T, X\rangle$ s.t.:*

- *P is a set of ports;*
- *L is a set of control locations;*
- *X is a set of variables such that $\bigcup_{p\in P} p.vars \subseteq X$;*
- *$T = L \times P \times \mathbb{B}[X] \times \mathcal{F} \times L$ is the set of transitions, where $\mathbb{B}[X]$ (resp. $\mathcal{F}$) is the set of boolean predicates (resp. update functions) over X.*

In a transition $\tau = \langle \ell, p_\tau, g_\tau, F_\tau, \ell'\rangle \in T$, (1) ℓ is the source location; (2) ℓ' is the destination location; (3) p_τ is the port exported by the component; (4) g_τ is the guard (a boolean predicate), a boolean function over X; (5) F_τ is an update function over X. For a component $B = \langle P, L, T, X\rangle$, we denote P, L, T, X, by $B.locs$, $B.ports$, $B.trans$, $B.vars$, respectively. Additionally, we denote by $\mathcal{B}$ the set of all atomic components. Furthermore, for a transition $\tau = \langle \ell, p, g, F, \ell'\rangle$, we denote ℓ, p, g, F, ℓ' by $\tau.src$, $\tau.port$, $\tau.guard$, $\tau.func$, $\tau.dest$, respectively. We denote by $\mathrm{readguard}(g_\tau)$ the set of variables appearing in the expression defining g_τ.

The semantics of an atomic component B is defined as an LTS. A state of the LTS consists of a location ℓ and valuation v of the variables of B where a valuation is a function from the variables of the component to a set of values. First, we define how an update function modifies a valuation. For an update function F and a valuation v, executing F on v yields a new valuation v', noted $v' = F(v)$, such that v' is obtained in the usual way by the successive applications of the assignments in F taken in order and where the right-hand side expressions are evaluated with the latest constructed temporary valuation. Moreover, for two valuations v and v', v'/v denotes the valuation where values in v' have priority over those in v.

A transition $\langle \ell, p[X_p], g_\tau, F_\tau, \ell'\rangle$ is possible to a new state $\langle \ell', v'\rangle$ iff B has a transition $\tau = \langle \ell, p[X_p], g_\tau, F_\tau, \ell'\rangle \in B.trans$ such that: (1) the guard before receiving the new valuation v_p

of the port variables holds, i.e., $g_\tau(v) = \texttt{true}$, and (2) the application of the update function $F_\tau(v_p/v)$ yields v'. A transition is labeled with port along with valuation of its variables v_p, which is possibly received from other components.

Definition 4 (Semantics of an atomic component). *The semantics of an atomic component B is the LTS $S_B = \langle B.locs \times \boldsymbol{X}, B.ports \times \boldsymbol{X}, \rightarrow \rangle$, where:*
$\rightarrow = \{\langle \langle \ell, v \rangle, p(v_p), \langle \ell', v' \rangle \rangle \mid \exists \tau = \langle \ell, p[X_p], g_\tau, F_\tau, \ell' \rangle \in T.trans : g_\tau(v) \wedge v' = F_\tau(v_p/v)\}$*; and, $\boldsymbol{X}$ denotes the set of possible valuations of the variables in $B.locs$.*

Furthermore, we say that a port p is enabled in a state $\langle \ell, v \rangle$, if there exists at least one transition τ from ℓ labeled by p and its guard $g_\tau(v)$ holds.

Interactions serve as the glue that coordinates (i.e., synchronization and data transfer) the components through their ports. We consider $\mathcal{B} = \{B_1, \ldots, B_n\}$ a set of atomic components where the semantics of B_i is $S_{Bi} = \langle Q_{Bi}, P_{Bi}, \rightarrow \rangle$, $i \in [1, n]$, and a set of interactions γ. An interaction consists of one or more ports of different atomic components, a guard on the variables of its ports, an update function that realizes data transfer between the ports.

Definition 5 (Interaction). *An interaction $a \in \gamma$ is a tuple $\langle P_a, F_a, G_a \rangle$ s.t.:*

- *$P_a \subseteq \bigcup_{B \in \mathcal{B}} (B.ports)$ is a set of ports including at most one port per atomic component, i.e., $\forall B \in \mathcal{B} : |B.ports \cap P_a| \leq 1$.*
- *F_a is an update function executed with the interaction such that*

$$(\text{readvar}(F_a) \cup \text{writevar}(F_a)) \subseteq \bigcup_{p_i \in P_a} (p_i.vars).$$

- *g_a is a boolean expression, the guard of the interaction.*

For an interaction a, we denote P_a, g_a, F_a, as $a.ports$, $a.guard$, $a.func$, respectively.

A composite component is defined by composing atomic components using glue consisting of interactions and priorities.

Definition 6 (Semantics of a composite component). *The semantics of the composite component built with $\mathcal{B}$ and γ (noted $\gamma(\mathcal{B})$) is the LTS $\langle Q, \gamma, \rightarrow \rangle$ where $Q = Q_{B_1} \times Q_{B_2} \times \ldots \times Q_{B_n}$, and $\rightarrow$ is the least set of transitions satisfying the following rule:*

$$\frac{I \subseteq [1,n] \qquad a = (\{p_i\}_{i \in I}, g_a, F_a) \in \gamma \qquad g_a(\{v_{p_i}\}_{i \in I}) \qquad \forall i \in I : q_i \xrightarrow{p_i(v_i)}_i q_i' \wedge v_i = g_a^i(\{v_{p_i}\}_{i \in I}) \qquad \forall i \notin I : q_i = q_i'}{\langle q_1, \ldots, q_n \rangle \xrightarrow{a} \langle q_1', \ldots, q_n' \rangle}$$

where v_{p_i} is the valuation of the variables attached to port p_i and F_a^i is the partial update function derived from F_a restricted to the variables of p_i.

An interaction a is enabled iff its guard g_a holds and all of its ports are enabled. An enabled interaction is selected from all interactions, based on the current states of the atomic components. The BIP engine selects one of the enabled interactions and executes its update function F_a, which may modify its port variables. Then, the involved atomic components (with indices in set

I) execute their corresponding transitions given the new valuations v_i received by the selected ports. In the following, we consider a composite component $\mathcal{C} = \gamma(\mathcal{B})$ with behavior $\langle Q, \gamma, \rightarrow \rangle$.

Multiple interactions can be enabled in a configuration. Priorities are used to filter the enabled interactions and reduce non-determinism.

Definition 7 (Priority). *A priority model π over $\mathcal{C}$ is a strict partial order on the set of interactions γ. We abbreviate $\langle a, a' \rangle \in \pi$ by $a \prec_\pi a'$. Adding π to $\mathcal{C}$ results in a new composite component $\mathcal{C}' = \pi(\mathcal{C})$ which semantics is the LTS $\langle Q, \gamma, \rightarrow_\pi \rangle$ where $\rightarrow_\pi$ is the least set of transitions satisfying the following rule:*

$$\frac{q \xrightarrow{a} q' \qquad \neg(\exists a' \in \gamma, \exists q'' \in Q : a \prec_\pi a' \wedge q \xrightarrow{a'} q'')}{q \xrightarrow{a}_\pi q'}$$

Whenever according to π an interaction $a \in \gamma$ is selected, there does not exist an enabled interaction in γ which has higher priority than a.

A composite component, obtained by the composition of a set of atomic components, can be composed with other components (composite or atomic) in a hierarchical and incremental fashion using the same operational semantics. Our method can be applied to a hierarchical system using two different approaches: (1) by flattening the system into one composite component, and (2) using scoping rules to target individual composite components. Without loss of generality, in the first approach, we flatten a hierarchical composite component to obtain a non-hierarchical one (i.e., consisting only of atomic components and simple interactions) using the method presented in [10]. The method ensures the existence of a mapping between the hierarchical to the non-hierarchical component: all transformations applied to the flattened component can be mapped to the original system. The non-hierarchical composite component resulting from the flattening is subsequently referred to as the *BIP model*. The BIP model is a single closed composite component, it has no ports that are exposed to an external entity. The second approach uses syntax directives to determine the scope of the transformations. A composite component is a combination of other composite components or atomic components, both behave similarly at the interactions level (i.e., they have a similar interface through ports). As such, it is possible to simply restrict the scope to a given composite component, and treat it as a global system, using scoping syntax to determine the targeted interactions at a given level in the hierarchy. In this approach, a composite component is seen as a grey box, where the interactions and interfaces of components that form it are available, and one can iterate over the sub-components as necessary. A BIP system is the instantiation of a BIP model, it defines the initial locations and variable initialization of atomic components.

Definition 8 (BIP system). *A BIP system is a tuple $\langle \mathcal{C}, q_0 \rangle$, where $q_0 = \langle Init, v \rangle$ is the initial state with $Init \in B_1.locs \times \ldots \times B_n.locs$ being the tuple of initial locations of atomic components, and $v \in \boldsymbol{X}^{Init}$ is the tuple formed by the initial valuations of all variables in atomic components $X^{Init} \subseteq \bigcup_{B \in \mathcal{B}}(B.vars)$.*

Example 1 (BIP system). *Figure 1 depicts a BIP system composed of two atomic components:* `Ping` *and* `Pong`. *The* `Ping` *component has one variable x_1 initialized to a random number and two locations* $\texttt{IDL}_1$ *and* `SND`*, and two ports* $\texttt{send}_1$ *and* $\texttt{recv}_1$. *We associate the variable x_1 with both* $\texttt{send}_1$ *and* $\texttt{recv}_1$. *Initially, the* `Ping` *and* `Pong` *components are at the* $\texttt{IDL}_1$ *and* $\texttt{IDL}_2$ *locations, respectively. From location* $\texttt{IDL}_1$*, in component* `Ping`*, port* $\texttt{send}_1$ *is enabled,*

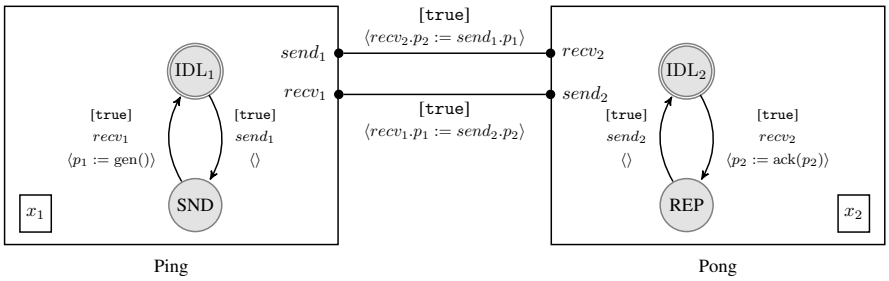


Figure 1: Two communicating agents

since the guard of the transition from $\texttt{IDL}_1$ *to* SND *holds. Similarly the transition from* $\texttt{IDL}_2$ *to* REP *in* Pong *is possible, and* $\texttt{recv}_2$ *is enabled. The interaction that has both ports* $\texttt{send}_1$ *and* $\texttt{recv}_2$ *enabled, and its guard holding, is now enabled. Since no other interaction is enabled, it executes. Its update function executes the data transfer using the ports* $\texttt{send}_1$ *and* $\texttt{recv}_2$ *and their associated variables* x_1 *and* x_2*. Then, the update function of each transition executes, generating the acknowledgment packet in* Pong*.* Ping *will move to location* SND *and* Pong *to* REP*. Similarly, on the next step, the acknowledgement is sent back to* Ping *and it generates a new number. The two interactions ensure synchronization between the two components.*

We abstract the execution of a BIP system as a BIP trace. The trace of a BIP system consists of the state changes and interactions taken.

Definition 9 (BIP trace). *A BIP trace* $\rho = (q_0 \cdot a_0 \cdot q_1 \cdot a_1 \cdots q_{i-1} \cdot a_{i-1} \cdot q_i)$ *is an alternating sequence of states of* S *and interactions in* γ*; where* $q_k \xrightarrow{a_k} q_{k+1} \in \rightarrow$*, for* $k \in [0, i-1]$.

The trace of a BIP system consists of the state changes and interactions executed.

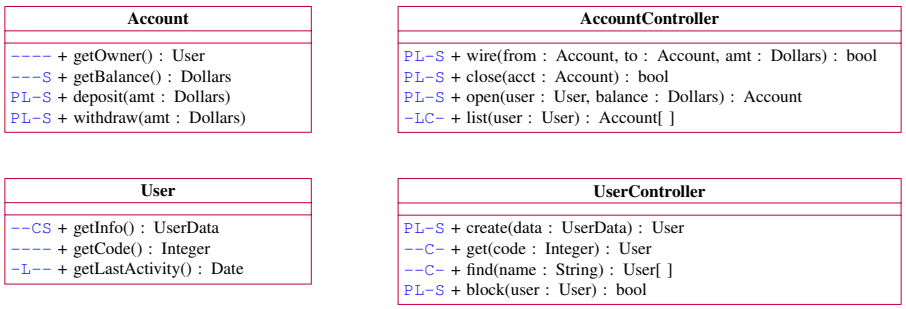
Example 2 (BIP Trace). *Recall the system presented in Example 1 consisting in the two components* Ping *and* Pong*. The first change in the state results in the following trace:*

$$\langle\langle \texttt{IDL}_1, \langle 1\rangle\rangle, \langle \texttt{IDL}_2, \langle 0\rangle\rangle\rangle \cdot \{\texttt{send}_1, \texttt{recv}_2\} \cdot \langle\langle \texttt{SND}, \langle 1\rangle\rangle, \langle \texttt{REP}, \langle 2\rangle\rangle\rangle$$

The above trace shows the alternations of locations and valuations of the components, and the interaction that is only represented by its ports for clarity. Initially Ping *is in* $\texttt{IDL}_1$ *and* Pong *is in* $\texttt{IDL}_2$*, with the variables* x_1 *and* x_2 *being initialized to* 1 *and* 0 *respectively. We use a shorthand of the interaction for clarity referring only the ports, in which case it is the interaction that consists of the set* $\{\texttt{send}_1, \texttt{recv}_2\}$*. Then, we show the resulting state after executing the interaction.* Ping *(resp.* Pong*) moves to the location* SND *(resp.* REP*), and the values for* x_1 *and* x_2 *are respectively* 1 *and* 2*. The value of* p_2 *has been first set to* 1 *through the interaction, then to* 2 *as the result of applying* $\mathrm{ack}(1)$ *in the transition.*

3. Aspect-Oriented Programming (AOP)

The implementation of crosscutting concerns mentioned in the introduction leads to two typical problems: *scattering* and *tangling* [37].



P: Persistence L: Logging C: Caching S: Security Policy

Figure 2: Multiple concerns in a simple system

- *Tangling* happens when concerns overlap in one region of the program. Consequently, enforcing one concern may affect others.
- *Scattering* is the dual situation of tangling. It happens when one concern is spread across different regions of the program. Scattering concerns go against encapsulation. Developers have to manually keep track of the location of a specific concern in multiple areas of the system.

In the following example, we illustrate the above two problems on an example.

Example 3 (Crosscutting concerns). *Figure 2 illustrates four different: logging, caching, persistence and security policy. A class diagram describing the classes' main methods is presented, we omitted to describe their relationships for clarity. The class diagram methods are prefixed with the four concerns as flags. If a method has a concern then some code related to the logic of that concern is included in the method. For example, the method* `Account.withdraw` *has three* tangled *concerns: persistence, logging, and policy. Thus, the method* `withdraw` *has to include code for persistence, logging, and logic. This code enforces the policy in addition to its own main logic. The policy concern is scattered across all four classes, hence maintaining it requires one to modify all four classes when a change is needed.*

The purpose of Aspect-Oriented Programming (AOP) is to localize crosscutting concerns in an aspect. An aspect is defined in [33] as "*a well-modularized implementation of a crosscutting concern*". These concerns are separated from the main program logic and contained in separate logical units. One example of separation of concerns is achieved by AspectJ [33], which is an aspect-oriented extension to the Java programming language. A *joinpoint* is a well-defined point in program execution where a concern needs to be handled. It acts as a reference point to coordinate the behavior of multiple concerns. A *pointcut* refers to a set of *joinpoints* and execution context information. Basic pointcuts can be composed and identified so as to increase re-usability. Pointcuts are the syntactic elements used to select joinpoints. A pointcut specifies a function signature, a variable name, and a module that needs to be matched. Furthermore, pointcuts are able to specify dynamic execution constraints, such as a function being invoked while inside another function (e.g., `cflow` pointcut in AspectJ). A pointcut regulates scattering by describing the joinpoints needed to implement the concern. An *advice* defines the additional behavior to be executed at each specific joinpoint selected by a pointcut. An *aspect* serves as the modular unit that encapsulate advices, pointcuts, and additional behavior. Furthermore, aspects may introduce

their own variables, methods, and fields. This is referred to as inter-type declarations. The term inter-type designates the fact that these extra objects and code are accessible in different types (based on the matching joinpoints). The main task of an AOP language implementation is to coordinate the execution of the non-aspect code with the aspect code. This coordination has to ensure a correct execution at the joinpoint of both primary and secondary concerns. This process is called *weaving* and can be done at compile-time, load-time, or run-time.

Listing 1: Logging concern

```
1  public aspect Logging {
2          private static Logger logger = Logger.getLogger(Logging.class.getName());
3          pointcut log() :
4          call(void Account.deposit(Dollars))
5                  || call(void Account.withdraw(Dollars))
6                  || call(* AccountController.*(*))
7                  || call(Date User.getLastActivity())
8                  || call(User UserController.create(UserData))
9                  || call(bool UserController.block(User)
10
11         after() returning(Object res) : log() {
12                 logger.info(thisJoinPoint.getSignature().toShortString()
13                         + Arrays.toString(thisJoinPoint.getArgs())
14                         + " -> " + res);
15         }
16 }
```

Example 4 (Logging concern). *Listing 1 implements parts of the logging concern shown in Example 3 using AspectJ. In the case of logging, the inter-type variable is a* `Logger` *object (Line 2). The pointcut expression (lines 4-9) specifies the various method invocations to be intercepted and names the pointcut* `log`*. The advice implements the logging code. It consists of code necessary to (i) capture the arguments of the method invoked using the* `getArgs()` *method on the* `thisJoinPoint` *object (ii) capture the return value of the method invoked, and (iii) pass it to the logger. The advice is set to trigger after the pointcut (line 11), in which case, it means after a method returns. Effectively, for any of the methods defined in the pointcut, the logger will log the name of the method called, its arguments and return value.*

4. Overview and Preliminaries of Aspect-Oriented Programming for CBSs

Defining execution points for crosscutting concerns in a program implicitly relies on the programming paradigm. In the case of AspectJ, the pointcuts are described using object-oriented terminology. For example, a pointcut is defined by specifying object types, their methods (messages) with their arguments and return values, and the fields accesses of a given object. As such the crosscutting concerns are described using object-oriented terminology. To illustrate this point, consider the fact that it is not possible to intercept changes to local variables inside methods using AspectJ because of the encapsulation and data-hiding concepts that are specific to object-oriented design. Similarly, when designing AOP pointcuts for CBSs, we restrict the pointcuts to what is relevant to component-based design.

4.1. Component-Based AOP

Developing component-based systems is a process of progressively repeating the following two stages. The first stage consists in building components that follow a certain interface. The interface defines the behavior the component must implement. In this stage, a component is a white-box, its internals are exposed to the component itself. The second stage, components are composed using their interfaces to form a system (e.g. an architecture). In this stage, a

component is a black-box, its internals are not visible to the system, only its interface is. As such for CBSs in general, it is important to handle crosscutting concerns that may arise in both stages. Therefore, one can distinguish two views of CBSs, *the local view and the global view*. The local view is concerned with the component design itself (i.e. the first stage), and the global view is concerned with the interaction of components (i.e. the second stage).

The local view and global view are independent, in the sense that components must only implement interfaces but they can do so separately. This separation effectively characterizes the joinpoints that one can reason about. In the local view, we consider the state of the component itself and actions that modify the state. In the global view, we examine the interaction between components by examining the passing of messages across the interfaces and synchronization.

Our notion of advice includes additional computation that executes before and after a given joinpoint. In the case of local advices, we include the ability to change the location of the component. The advice can require storage of additional state information. As such, intertype variables are required for both the local and global views. They need to be accessible for the advice and each of the levels separately. Once woven into the system, advices can modify the system behavior. While it is possible to modify the system arbitrarily, we define our notion of correctness. In the scope of this paper, we consider the correctness of advice *application*, i.e., we verify the advice has been placed correctly at the matched joinpoint.

4.2. Overview of AOP for BIP

The BIP framework is endowed with formal semantics that describes both views. Atomic components (Definition 3) describe the internals of a given component and its implementation, while the BIP model (Definition 8) formalizes the composition of atomic components and their priorities. We recall the distinction between three various notions. First, the BIP model consists of the elements needed to represent a given system. It is used to form the LTS that represents the behavior of the system. Second, the BIP system is a runtime instantiation of the model. And lastly, the BIP trace contains the elements of the model that are executed by the LTS during runtime. While the global BIP trace (Definition 9) contains all information, we consider restrictions so that we separate the internals of components from the interactions between components. Using the information from each view, we define two types of joinpoints: global and local.

Global joinpoints. In the global view, concerns are at the level of interactions. Therefore, we focus on the interface of components. This view evaluates concerns that crosscut interactions (i.e., ports, synchronization and data transfer). The components export only their ports, on which interactions are defined. Generally, each component computes its enabled ports. The interactions that have all their ports enabled and their guard evaluating to `true`, are said to be enabled. The system executes the enabled interaction with the highest priority. Therefore, at the interaction level, the following operations exist: interaction enablement and interaction execution. We do not consider interaction enablement, since to inspect and instrument around it requires knowledge of the internals of components. Therefore, it is not compatible with the global view. Whenever an interaction executes, three kinds of global joinpoints can be identified: (1) synchronization between different atomic components (2) one or more atomic components sending data (3) one or more atomic components receiving data. These three joinpoints are captured by the interaction: the ports define synchronization, and variables read or written define data transfer. From a BIP trace (Definition 9), one can extract the sequence of executed interactions called the global trace.

Definition 10 (Global Trace). *The global trace extracted from the BIP trace* $\rho = (q_0 \cdot a_0 \cdot q_1 \cdots a_{i-1} \cdot q_{i-1} \cdot a_{i-1} \cdot q_i)$*, noted* $T(\rho)$*, is* $(a_0 \cdot a_1 \cdots a_{i-1})$.

A global joinpoint is an interaction execution moving the system from a state to another state. We suppress the state information from the BIP trace, as the atomic components are blackboxes for the global view. For the rest of the paper, we consider $\mathcal{E}$ to be the set of all possible interaction executions in $\langle \mathcal{C}, q_0 \rangle$ with $\rightarrow$, and fix an arbitrary BIP trace ρ. We use two sets γ and $\mathcal{E}$ to distinguish between the *syntax* representing the interactions, and the actually *executed* interaction, respectively.

Local joinpoints. In the local view, we focus on atomic components seen as white boxes and seek to refine their behavior. We study the state of an atomic component to locate possible points where crosscutting concerns apply. The behavior of the atomic component consists of an LTS that changes states when a transition is taken. Therefore, we consider concerns are at the level of locations, states, transitions, guards and computations on transitions. In order to study the local view, we must define first how an atomic component contributes to the global execution of the BIP system. Since in this view we see components as white boxes, we have knowledge of the full BIP system and can extract a local execution trace for a given atomic component. We fix an atomic component B_k with semantics defined by the LTS $S_{B_k} = \langle Q_{B_k}, P_{B_k}, \rightarrow_{B_k} \rangle$. In order to define events local to a given atomic component, we first define the projection of the global state on a local component to extract its own state.

Definition 11 (State projection). *The projection of a global state $q = \langle q_0, \ldots q_n \rangle$ on a local component B_k is defined as $q \upharpoonright_{B_k} = q_k$.*

An event local to a component is defined as a triple $\langle \langle l, v \rangle, \tau, \langle l', v' \rangle \rangle$. This event denotes that an atomic component has moved from the state with location l and valuations v to another state with location l' and valuations v' using a transition τ. Given a composite component $\mathcal{C}$ composed of atomic components in a set $\mathcal{B} = \{B_1, \ldots, B_n\}$, and its BIP trace $\rho = (q_0 \cdot a_0 \cdot q_1 \cdot a_1 \cdots q_{i-1} \cdot a_{i-1} \cdot q_i)$, we say that $E_i = \langle q_i, a_i, q_{i+1} \rangle$ is a global event. A global event represents a global move in the BIP system from a state to another, after executing an interaction.

Definition 12 (Event projection). *We project a global event E_i to a local event e_i in the atomic component B_k by using* $\mathrm{map}(E_i, B_k)$ *where:*

$$\mathrm{map}(E_i, B_k) = \begin{cases} \langle \langle l_i, v_i \rangle, \tau, \langle l_{i+1}, v_{i+1} \rangle \rangle & \textit{if } \langle l_i, v_i \rangle = q_i \upharpoonright B_k \\ & \wedge \langle l_{i+1}, v_{i+1} \rangle = q_{i+1} \upharpoonright B_k \wedge p = \tau.\textit{port} \\ & \wedge\, p \in a_i.\textit{ports} \wedge \tau.\textit{guard}(v_i) \\ & \wedge\, v_p = F_a^p(\{v_{p_j}\}_{p_j \in a.\textit{ports}}) \\ & \wedge\, v_{i+1} = \tau.\textit{func}(v_p / v_i) \\ \epsilon & \textit{otherwise} \end{cases}$$

The map function searches within the interaction a_i executing globally for any ports in the atomic component that are both enabled and included in a_i. If no ports are found, then other components are involved in a and thus the global event does not concern the local component (and map returns ϵ). Otherwise, the function map projects both the global state before and after the interaction to the local component and set τ to be the interaction that enabled the port for the interaction to execute, and takes the local component from $\langle l_i, v_i \rangle$ to $\langle l_{i+1}, v_{i+1} \rangle$. For a local event $e_i = \langle \langle l_i, v_i \rangle, \tau, \langle l_{i+1}, v_{i+1} \rangle \rangle$ we denote $\langle l_i, v_i \rangle, \langle l_{i+1}, v_{i+1} \rangle, l_i, l_{i+1}, v_i, v_{i+1}, \tau$ by e.q, e.q$'$, e.l, e.l$'$, e.v, e.v$'$, and e.τ, respectively. We then extend function map to a sequence of global events: $\mathrm{map}(E_0 \cdots E_i, B_k) = \mathrm{map}(E_0, B_k) \cdots \mathrm{map}(E_i, B_k)$, where ϵ is interpreted as the neutral element of concatenation (i.e., $E \cdot \epsilon = \epsilon \cdot E = E$). In the sequel, we denote T_k the local trace of an atomic component B_k.

Table 1: Traces According to Views

BIP Trace (ρ)	$\langle\langle \texttt{IDL}_1, \langle 1\rangle\rangle, \langle \texttt{IDL}_2, \langle 0\rangle\rangle\rangle \cdot \{\texttt{send}_1, \texttt{recv}_2\} \cdot \langle\langle \texttt{SND}, \langle 1\rangle\rangle, \langle \texttt{REP}, \langle 2\rangle\rangle\rangle$
Global Trace (T)	$\{\texttt{send}_1, \texttt{recv}_2\}$
Local Trace (T_0)	$\langle \texttt{IDL}_1, \langle 1\rangle\rangle \cdot \texttt{send}_1 \cdot \langle \texttt{SND}, \langle 1\rangle\rangle$

Example 5 (Traces and views). *Table 1 shows traces associated with the views. We use the ports as shorthands to interactions and transitions for clarity. The BIP trace from Example 2 is presented in the first row. We recall the two atomic components* $B_0 = \texttt{Ping}$ *and* $B_1 = \texttt{Pong}$*. In the second row, the global trace associated with the global view is extracted from the BIP trace, it only contains the interactions executed, in this case* $\{\texttt{send}_1, \texttt{recv}_2\}$*. From the BIP trace, we extract the following global event:*

$$E_0 = \langle\langle \texttt{IDL}_1, \langle 1\rangle\rangle, \langle \texttt{IDL}_2, \langle 0\rangle\rangle\rangle, \{\texttt{send}_1, \texttt{recv}_2\}, \langle\langle \texttt{SND}, \langle 1\rangle\rangle, \langle \texttt{REP}, \langle 2\rangle\rangle\rangle\rangle$$

The third row shows the local trace for the Ping *component (*B_0*). It is obtained as follows:* $\mathrm{map}(E_0, B_0) = \langle \texttt{IDL}_1, \langle 1\rangle\rangle \cdot \texttt{send}_1 \cdot \langle \texttt{SND}, \langle 1\rangle\rangle$*. From the BIP trace, we extract only elements relevant to* B_0*. We know that the interaction* $\{\texttt{send}_1, \texttt{recv}_2\}$ *is executed, which is associated with the transition with port* $\texttt{send}_1$ *in* B_0*.*

To handle the concerns arising at both the local and global views, we need to formally identify and select the joinpoints described for each view. We leverage the operational semantics of a BIP model to associate joinpoints, pointcuts, and advices with the original model. For the rest of the paper, we fix an arbitrary BIP-system $\langle \mathcal{C}, q_0\rangle$ where $\mathcal{C} = \pi(\gamma(\mathcal{B}))$ is the BIP model with semantics $S = \langle Q, \gamma, \rightarrow\rangle$. In Section 5 and 6 we present the AOP concepts associated with the global and local views, respectively. For each view, we define an homogeneous notion of execution points in order to define where concerns can arise (i.e., joinpoints), and then the syntax to select these points (i.e., pointcuts). We recall that joinpoints represent points in the *execution* of a BIP system. Given a pointcut expression pc for a given view, we use the assertion "$\mathrm{e} \vDash \mathrm{pc}$" to indicate that an element e of a (local or global) trace matches pointcut pc. We define the legal actions at these execution points (i.e., advices) and how these actions are integrated into an existing model (i.e., weaving). We allow advices to store additional state information through the creation of additional variables both at the local and global view. These additional variables constitute the inter-type variables. Furthermore for each view, we define the function select, which returns the elements of the *model* (syntax) that are required to instrument so that in the resulting system, whenever the assertion ($\mathrm{e} \vDash \mathrm{pc}$) holds during runtime, we execute the advice.

4.3. Applicability to other CBS Frameworks

While we use the BIP semantics for implementation, we note that our approach applies to other CBS frameworks. In general, the program execution is abstracted as an execution trace on which AOP defines its matches and changes. Given the two defined views, we require a general trace of the system that provides information about the interactions between components. Then, knowing the internals of the system, we can project the global trace to a local trace, to determine the execution of the local component. Verifying the correctness of AOP for CBS is done by analyzing the traces and ensuring that the joinpoints match the correct trace fragments, and that advices modify the traces accordingly and minimally (i.e., only when the joinpoints match). While our approach relies on the BIP semantics for the instrumentation of aspects, one could

adapt it to any formal framework that defines the semantics of the behavior of a component, and the semantics of the interactions between components.

5. Global Aspects

Following the general requirements of the AOP approach for CBSs described in Section 4.2, we now address the concerns arising in the global view, namely the view where components are black boxes and only the interactions are visible. For this section, we consider the composite component $\mathcal{C} = \pi(\gamma(\mathcal{B}))$ as the BIP model with a BIP trace ρ.

5.1. Global Pointcuts

Since we consider only the interaction execution joinpoint, we consider criteria for matching interactions and relate them to global joinpoints. To select a set of interactions, we use constraints over their associated ports (and their variables) and the involved data transfer. For this, a global pointcut expression has three parts: the ports themselves, a set of read variables, and a set of write variables. Note that the port variables should be involved in the computation function of the interaction. In Section 5.2, we use the read and written variables to define the context information passed to the advice.

Definition 13 (Global pointcut). *A global pointcut is a 3-tuple $\langle p, v_r, v_w \rangle$ s.t.:*

- $p \subseteq \bigcup_{B \in \mathcal{B}}(B.ports)$ *is a set of ports,*
- $v_r \subseteq \bigcup_{p_i \in p}(p_i.vars)$ *is the set of* read *variables, and*
- $v_w \subseteq \bigcup_{p_i \in p}(p_i.vars)$ *is the set of* modified *variables.*

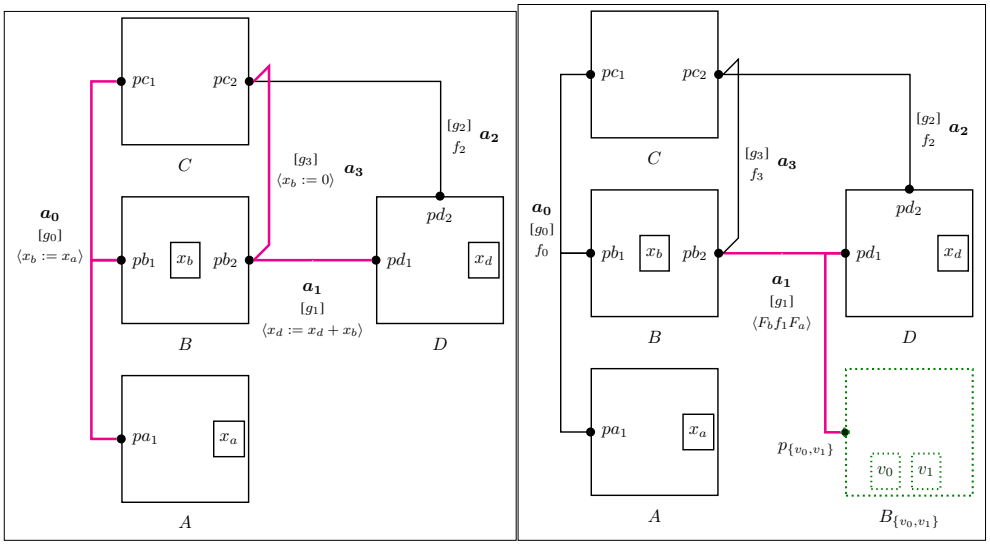
We recall from Definition 10 that the global trace is a sequence of executed interactions $(a_0 \cdot a_1 \cdots a_{i-1})$. Using the structure of an interaction (syntax only), we can match synchronization and data transfer. Synchronization is matched using ports and data transfer by syntactic analysis of variables read and written.

Definition 14 (Matching an executed interaction with a global pointcut). *An executed interaction a is a joinpoint selected by a global pointcut $\langle p, v_r, v_w \rangle$ iff $a \vDash \langle p, v_r, v_w \rangle$, where:*
$a \vDash \langle p, v_r, v_w \rangle$ *iff* $p \subseteq a.ports \wedge v_r \subseteq \mathrm{readvar}(a.func) \wedge v_w \subseteq \mathrm{writevar}(a.func)$.

By construction of the global trace, the executed interaction is the same interaction found in the model. As such, one can immediately match and modify the executed interactions by modifying the model directly. Matching a global pointcut consists in selecting a subset of the interactions of a composite component.

Proposition 1. $\forall a \in \mathcal{E} : a \vDash gpc$ *iff* $a \in \mathrm{select_g}(\mathcal{C}, gpc)$, *where:* $\mathrm{select_g}(\mathcal{C}, \langle p, v_r, v_w \rangle) = \{a' \in \gamma \mid p \subseteq a'.ports \wedge v_r \subseteq \mathrm{readvar}(a'.func) \wedge v_w \subseteq \mathrm{writevar}(a'.func)\}$

An interaction $a \in \gamma$ is selected by a global pointcut $\langle p, v_r, v_w \rangle$ if and only if a involves all the ports in p, and its update function reads from the variables in v_r and writes to the variables in v_w. The proposition states that during the execution, the interaction a is matched (i.e., $a \vDash gpc$)) iff the interaction a is syntactically selected (i.e., $a \in \mathrm{select_g}(\mathcal{C}, gpc)$).



(a) Matching (b) Weaving

Figure 3: Matching and Weaving a Global Aspect

Example 6 (Interactions matched by a pointcut). *Figure 3a shows the interactions obtained by matching four pointcuts:*

1. $\langle \{pa_1, pb_1\}, \emptyset, \emptyset \rangle$ *selects all interactions including* $\{pa_1, pb_1\}$ *in their ports, i.e., it selects only* a_0 *(e.g.* $\text{select}_{\text{g}}(\mathcal{C}, \langle \{pa_1, pb_1\}, \emptyset, \emptyset \rangle) = a_0$*) as it is the only interaction involving both ports.*

2. $\langle \{pb_2\}, \emptyset, \emptyset \rangle$ *selects all interactions including* $\{pb_2\}$ *in their ports, i.e., it selects interactions* a_1 *and* a_3*, since they both involve* pb_2*.*

3. $\langle \{pb_2\}, \{x_b\}, \emptyset \rangle$ *selects interactions including* $\{pb_2\}$ *and which computation reads variable* x_b *associated with* pb_2*. The pointcut only selects* a_1*.*

4. $\langle \{pd_1\}, \{x_d\}, \{x_d\} \rangle$ *selects interactions that include* $\{pd_1\}$ *and which computation read and write the variable* x_d *associated with* pd_1 *(to receive data). The pointcut only selects* a_1*.*

5.2. Global Advice and Global Aspect

A global advice defines the possible actions allowed on a global joinpoint $\langle q, a, q' \rangle$. These actions are restricted to two update functions F_b and F_a executed respectively before and after the interaction function $a.func$. Moreover, a global advice can modify only the ports that it matches, as interactions could include other ports. Normally an update function of the interaction can modify the variables of all ports. The non-matching ports and their variables are hidden from the advice as per application of Demeter's law [36], as such we assume as little as possible on the interactions, and promote loose-coupling. In the rest of the section, we consider a global pointcut $pc = \langle \{p_1, \ldots, p_n\}, v_r, v_w \rangle$, an advice is restricted to the set of ports $\{p_1, \ldots, p_n\}$ and their variables, and a set of extra variables V (inter-type variables).

Definition 15 (Global advice). *Given a set of ports $p \subseteq \bigcup_{B \in \mathcal{B}} B.ports$ and a set of inter-type variables V, $X_{\mathrm{adv}} = V \cup \bigcup_{p_i \in p}(p_i.vars)$ is the set of* advice variables*. A global advice is a pair of update functions $\langle F_b, F_a \rangle$ such that:*

- $(\mathrm{readvar}(F_b) \cup \mathrm{writevar}(F_b)) \subseteq X_{\mathrm{adv}}$*, and*
- $(\mathrm{readvar}(F_a) \cup \mathrm{writevar}(F_a)) \subseteq X_{\mathrm{adv}}$.

The global advice bound to p and V is noted $\mathrm{gadv}(p, V)$.

Functions F_b and F_a are referred to as the *before and after advice functions*, respectively. For simplicity and clarity, we assume the update functions F_b and F_a to be uniquely determined and not empty. To ensure this, one can add code markers at the start and end of each F_b and F_a which have no effect and are not present in the original system. The variables that they read and write (captured with $\mathrm{readvar}$ and $\mathrm{writevar}$, respectively) should be variables of the advice.

We bind an advice to a pointcut expression with a global aspect. The advice then applies to every joinpoint that the pointcut matches.

Definition 16 (Global aspect). *A global aspect is a tuple $\langle \mathcal{C}, V, gpc, \mathrm{gadv}(p, V) \rangle$ s.t.:*

- *$\mathcal{C}$ is a composite component;*
- *V is the set of inter-type variables associated with the aspect;*
- *$gpc = \langle p, v_r, v_w \rangle$ is the global pointcut (as per Definition 13);*
- *$\mathrm{gadv}(p, V)$ is the global advice (as per Definition 15).*

A global aspect therefore associates a pointcut to an advice. It ensures that the ports referred to the pointcut are the same for the advice, and that the advice has access to the variables of all ports in p and V.

Example 7 (Global advice and global aspect). *The global aspect*

$$\langle \mathcal{C}, \{v_0\}, \langle \{pd_1\}, \{x_d\}, \{x_d\} \rangle, \langle \langle v_0 := x_d \rangle, \langle x_d := v_0 \rangle \rangle \rangle$$

defines the inter-type variable v_0. It also defines the pointcut to match the interactions that include port pd_1 and which update function reads and writes to x_d. The before and after update functions associated with the advice are respectively $\langle v_0 := x_d \rangle$ and $\langle x_d := v_0 \rangle$; saving the value of x_d in v_0 before the update function executes and then setting it back afterwards. The pointcut matches a_1 as illustrated in Fig. 3a and specifies that a_1 must execute the following sequence of instructions: $\langle (v_0 := x_d), (x_d := x_d + x_b), (x_d := v_0) \rangle$. An advice function in this case can access only $\{v_0\} \cup pd_1.vars$. The advice functions have no access to x_b, as it is not related to port pd_1 but pb_2. This aspect disallows all interactions that read and write to x_d to modify its value.

5.3. Global Weaving

Using the global aspect, the weaving procedure instruments the BIP model. The procedure ensures that during the execution, the resulting BIP system will execute the advice whenever a joinpoint is reached. Recall that interactions are stateless (i.e., they have no variables of their own), but they rely on data transfer from ports. Variables can be defined only in atomic components. Therefore, the weaving procedure must create an extra atomic component (so called

inter-type component) that contains the variables of the aspect along with necessary ports and interactions. The weaving operation is concerned only with syntactically modifying the BIP model. For this purpose, we separate the two notions of matching to find the locations to modify from the instrumentation itself. We therefore define first the transformation procedure and then its application with matching.

The transformation procedure uses the following input parameters:

- a BIP composite component $\mathcal{C}$ (the BIP model);
- a set of interactions $\mathcal{I}$ resulting from matching with $\mathrm{select_g}$ (Proposition 1);
- a set of extra variables (i.e., the inter-type variables);
- the two functions F_b and F_a of the advice.

Accordingly, we create a new BIP composite component where the update function of each $a \in \mathcal{I}$ is preceded by F_b and followed by F_a. In the following, we describe the weaving of a global aspect which requires weaving of the inter-type component and weaving of the advice.

Generating an inter-type component. We first define the inter-type component.

Definition 17 (Inter-type component). *The inter-type component associated to the set of inter-type variables V is defined as $B_V = \langle \{p_V\}, \{\ell_0\}, \{\langle \ell_0, p_V, \texttt{true}, \langle\rangle, \ell_0 \rangle\}, V \rangle$ where $p_V = \langle p_V, V \rangle$.*

B_V contains V as its variables, one port $p_V = \langle p_V, V \rangle$ with all the variables attached to it, and one control location with a transition labeled with p_V and guarded with the expression `true`. This ensures that the port will not stop any other interaction from executing once connected to it. The inter-type component is added to the set of atomic components $\mathcal{B}$ of the BIP model.

Example 8 (Adding an inter-type component to a model). *Figure 3b depicts $\pi(\gamma(\mathcal{B} \cup \{B_V\}))$ where $V = \{v_0, v_1\}$ and $\mathcal{C} = \pi(\gamma(\mathcal{B}))$. A new atomic component B_V is created. B_V has the variables v_0 and v_1 and has its port p_V always enabled. The variables in V are attached to p_V.*

Weaving the advice. Once the inter-type component is added to the model, the advice is woven by connecting the existing interactions to it.

Definition 18 (Global weave). *Given a composite component $\mathcal{C} = \pi(\gamma(\mathcal{B}))$, a set of interactions $\mathcal{I}$, the set of inter-type variables V, and a global advice $adv = \langle F_b, F_a \rangle$, the global weave is defined as $\mathcal{C}' = \mathrm{weave_g}(\mathcal{C}, \mathcal{I}, V, adv)$ where $\mathcal{C}' = \pi'(\gamma'(\mathcal{B} \cup \{B_V\}))$ is the new composite component; with:*

- *$B_V = \langle \{p_V\}, \{\ell_0\}, \{\langle \ell_0, p_V, \texttt{true}, \langle\rangle, \ell_0 \rangle\}, V \rangle$ is the inter-type component identified by V (as per Definition 17);*
- *γ' is defined as $\{\mathrm{m}(a) \mid a \in \gamma\}$ with:*
  $\mathrm{m}(a) : \gamma \to \gamma' = \begin{cases} \langle a.ports \cup \{p_V\}, \langle F_b, a.func, F_a \rangle, a.guard \rangle & \text{if } a \in \mathcal{I}, \\ a & \text{otherwise.} \end{cases}$
- *$\pi' = \{\langle \mathrm{m}(a), \mathrm{m}(a') \rangle \mid \langle a, a' \rangle \in \pi\}$ is the updated priority model.*

Weaving a global aspect $GA = \langle \mathcal{C}, V, gpc, \langle F_b, F_a \rangle \rangle$ *into a composite component* $\mathcal{C}$ *is noted* $\mathcal{C} \triangleleft_g GA$ *and yields a new composite component* $\mathcal{C}' = \mathrm{weave_g}(\mathcal{C}, \mathrm{select_g}(\mathcal{C}, gpc), V, \langle F_b, F_a \rangle)$.

The inter-type component B_V is added to the model. The interactions that require instrumentation (i.e., those in $\gamma \cap \mathcal{I}$) are extended with the port p_V so as to have access to the inter-type variables and their computation function is prepended with F_b and F_a. The interactions not matched (i.e., those in $\gamma \setminus \mathcal{I}$) are unmodified and copied.

The priority model (π') is only extended to the changed interactions ($\mathrm{m}(a)$), but otherwise unmodified.

Example 9. *Figure 3b illustrates the weaving on the set of interactions* $\{a_1\}$ *with the set of inter-type variables* $V = \{v_0, v_1\}$ *of the advice* $\langle F_b, F_a \rangle$. *A new atomic component* B_V *is created that has two local variables* v_0 *and* v_1 *and has its port* p_V *always enabled. The variables in* V *are attached to* p_V.

- *Interaction* a_1 *is connected to* p_V *so as to allow access to* V *on which* F_b *and* F_a *can operate.*
- *Update function* F_b *is prepended to* $a_1.func$ *so as to execute before and* F_a *is appended to* $a_1.func$ *so as to execute after.*
- *Since* p_V *is always enabled, interaction* a_1 *will be enabled whenever* pb_2 *and* pd_1 *are both enabled and* g_1 *holds. The addition of* p_V *does not affect enablement.*
- *Once* a_1 *is executed,* F_b *and* F_a *can modify the inter-type variables in* B_V *by modifying* $p_V.vars$.

Correctness of weaving. Consider $\mathcal{E}'$ (resp. $\mathcal{E}$) to be the set of executed interactions in the output (resp. initial) BIP system $\langle \mathcal{C}', q_0' \rangle$ (resp. $\langle \mathcal{C}, q_0 \rangle$), where $\mathcal{C}' = \mathcal{C} \triangleleft_g \langle \mathcal{C}, V, gpc, \langle F_b, F_a \rangle \rangle$. We begin by defining function $\mathrm{rem_g} : \mathcal{E}' \times \mathcal{F} \times \mathcal{F} \to \mathcal{E} \cup \{\epsilon\}$. Function $\mathrm{rem_g}$ removes the global advice from an interaction in $\mathcal{E}'$ and constructs a similar interaction in $\mathcal{E}$ or the empty interaction ϵ if it does not match the advice.

$$\mathrm{rem_g}(a, F_b, F_a) = \begin{cases} a' & \text{if } \exists F : a.func = \langle F_b, F, F_a \rangle, \\ \epsilon & \text{otherwise,} \end{cases}$$

with: $a' = \langle a.ports \setminus \{p_V\}, F, a.guard \rangle$, where $a.func = \langle F_b, F, F_a \rangle$.

The following proposition expresses the correct application of the advice on the joinpoints selected by a pointcut expression.

Proposition 2 (Weaving correctness).
$\forall a \in \mathcal{E}', \exists F : a.func = \langle F_b, F, F_a \rangle$ *iff* $(e' \neq \epsilon \wedge e' \vDash gpc)$, *with* $e' = \mathrm{rem_g}(a, F_b, F_a)$.

We say that the update function of an interaction satisfies an advice weaving if its update function ($a.func$) starts with F_b and ends with F_a (i.e., the before and after update functions). Proposition 2 states that any interaction in the execution satisfies an advice weaving iff one can construct an event e' without the advice with F_b and F_a ($e' = \mathrm{rem_g}(a, F_b, F_a)$) which matches gpc ($e' \vDash gpc$) in the initial system. Since an advice can add extra behavior such as reading and writing to variables, it can cause the event to match more joinpoints, therefore we remove the extra update functions before matching with gpc.

Listing 2: Local Pointcut Expression

```
lpc  : lpc 'and' lpc
     | atom;
atom : 'atLocation' '(' ℓ ')'
     | 'readVarGuard' '(' x ')'
     | 'readVarFunc' '(' x ')'
     | 'write' '(' x ')'
     | 'portEnabled' '(' p ')'
     | 'portExecute' '(' p ')'
```

Example 10 (Correctness of global weave). *We use the example with components* `Ping` *and* `Pong`*, and its traces presented in Example 5. The global trace consists of the single interaction* $a = \langle \{\mathtt{send}_1, \mathtt{recv}_2\}, \langle recv_2.p_2 := send_1.p_1 \rangle, \mathtt{true} \rangle$*. Consider the global aspect* $\mathrm{GA} = \langle \mathcal{C}, \emptyset, \langle \{\mathtt{send}_1\}, \emptyset, \emptyset \rangle, \langle F_\mathrm{b}, F_\mathrm{a} \rangle \rangle$ *.* GA *executes the advice when an interaction with port* $\mathtt{send}_1$ *executes. The global trace of the system after weaving consists of the interaction*

$$a' = \langle \{\mathtt{send}_1, \mathtt{recv}_2\}, \langle F_\mathrm{b} \cdot recv_2.p_2 := send_1.p_1 \cdot F_\mathrm{a} \rangle, \mathtt{true} \rangle .$$

This is correct w.r.t Proposition 2: we have $\mathrm{rem}_\mathrm{g}(a', F_\mathrm{b}, F_\mathrm{a}) = a$*, and* $a \vDash \langle \{\mathtt{send}_1\}, \emptyset, \emptyset \rangle$ *holds. We note that the proposition ensures that* F_b *and* F_a *cannot occur in any interaction execution that does not match the pointcut. Furthermore, it is sufficient to directly edit* a *in the model so that the edit appears during the execution (Proposition 1).*

6. Local Aspects

After discussing concerns that arise at the global view and following the approach discussed in Section 4.2, we now address the concerns arising in the local view. An atomic component has control locations, variables, and transitions labeled with ports, guards and computation functions. For the local view, a component B_k "sees" a sequence of local events (Definition 12). A local event consists of a source state containing the location and variable valuations, a transition, and a new state. We define local joinpoints as local events, as they can be associated with the local execution of a component. We are now concerned with the matching of these joinpoints. We consider port execution and enablement, guard evaluation, access and modification of the state of the LTS (i.e., current location and valuation of local variables).

6.1. Local Pointcuts

A local pointcut expression selects local joinpoints. Considering the multitude of joinpoints, we use a grammar for specifying the pointcut expression, which makes it more readable and easier to combine. Pointcut expressions are defined using the grammar in Listing 2, where $\ell \in B_k.locs, x \in B_k.vars, p \in B_k.ports$.

Definition 19 (Matching of a local joinpoint with a local pointcut expression). *A local joinpoint* e *matches the pointcut expression* lpc*, noted* $e \vDash lpc$*, iff match* lpc *with*

$$
\begin{array}{ll}
\mid \phi \quad \text{and} \quad \phi' & \rightarrow e \vDash \phi \wedge e \vDash \phi' \\
\mid \mathrm{atLocation}(\ell) & \rightarrow (e.l = \ell) \\
\mid \mathrm{readVarGuard}(x) & \rightarrow (\exists t \in B_k.\mathit{trans} : t.\mathit{src} = e.l \wedge x \in \mathrm{readguard}(t)) \\
\mid \mathrm{readVarFunc}(x) & \rightarrow (x \in \mathrm{readvar}(e.\tau.\mathit{func})) \\
\mid \mathrm{write}(x) & \rightarrow (x \in \mathrm{writevar}(e.\tau.\mathit{func})) \\
\mid \mathrm{portEnabled}(p) & \rightarrow (\exists q' : \langle e.q, p, q' \rangle \in \rightarrow_{B_k}) \\
\mid \mathrm{portExecute}(p) & \rightarrow (e.\tau.\mathit{port} = p)
\end{array}
$$

A component is considered at a location ℓ (predicate $\mathrm{atLocation}(\ell)$) if the component produces an event containing ℓ as the starting location of the transition of a local event.

Remark 1 (Location). *While we can consider a component to be* $\mathrm{atLocation}(\ell)$ *when it produces an event containing* ℓ *as the ending location, we note that* $\mathrm{atLocation}(\ell)$ *includes evaluation of the guards and enabled ports. Therefore, we restrict* $\mathrm{atLocation}(\ell)$ *to select only events that start with* $\mathrm{atLocation}(\ell)$ *as they capture better that reasoning since that evaluation is done to decide the transition to use and the next location. As a convention, we assume that the component is at the location at the start of an event until the update function of the transition begins executing.*

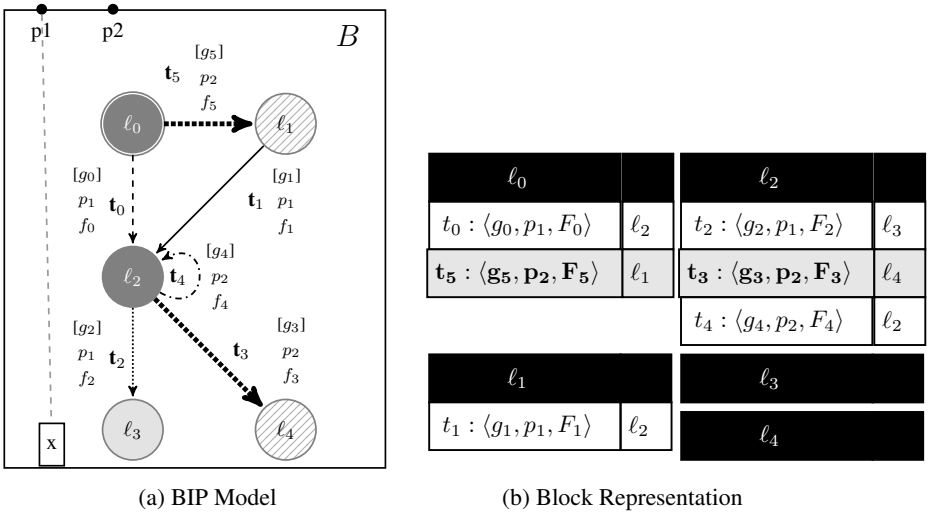
The evaluation of a variable x in the guard is checked on all outgoing transitions from a location regardless of the transition execution. Therefore, for $\mathrm{readVarGuard}(x)$, we check if any transition has x in its guard expression and originates from the location $e.l$. In contrast, an update function executes only when the transition is executed so we only examine the update function of $e.\tau$, and check if a variable x is read ($\mathrm{readVarFunc}(x)$) or modified ($\mathrm{write}(x)$). For port enablement ($\mathrm{portEnabled}(p)$), we check if there exists in the semantics of the component at least one transition labeled with the port p from the event start state to any other state. While for port execution ($\mathrm{portExecute}(p)$), we only compare against $e.\tau$ as we are interested in the port that will execute.

Remark 2 (Simplification). *Note that, from the semantics of BIP (Definition 6, p. 5), whenever* $\mathrm{portExecute}(p)$ *holds, then* $\mathrm{portEnabled}(p)$ *holds. Therefore, it is possible to simplify a local pointcut expression by replacing, for the same* p*, '*$\mathrm{portExecute}(p)$ *and* $\mathrm{portEnabled}(p)$*' by* $\mathrm{portExecute}(p)$.

6.2. Matching Pointcuts

Unlike global pointcuts which map directly to interactions, local pointcuts require more complex instrumentation (i.e., location, transitions, guards, update functions). Therefore, we group the syntactic elements and define which elements precede or succeed them. This allows us to relate syntactic elements (in the BIP model) to a notion of before and after during the execution of the BIP system.[1] We use two levels of granularity. The fine (resp. coarse) granularity focuses on a given transition (resp. a group of transitions grouped by source location). We refer to a group of transitions grouped by source location ℓ as the location block ℓ. Transitions are grouped into blocks since according to the BIP semantics (Definition 4), to move from a location to another,

[1] Similarly, in a regular program, adding extra code before a function call at the source-code level, results in executing the code before the function call during the execution of the program. However, we note that this is not as straightforward for the semantics of BIP.



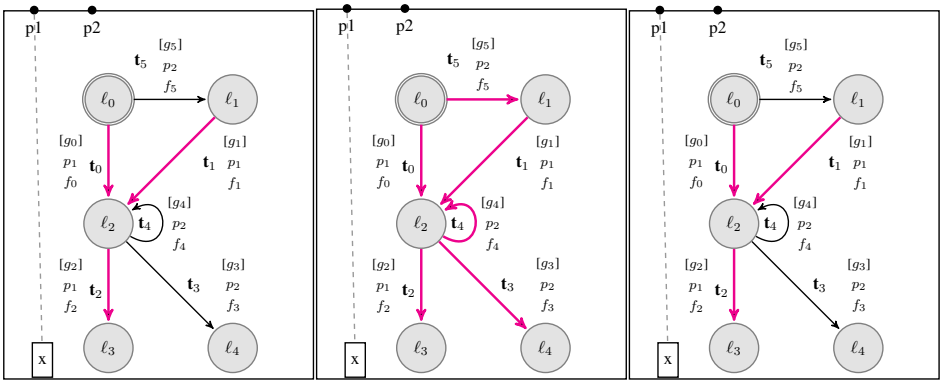
(a) BIP Model (b) Block Representation

Figure 4: Syntax Representations

it is necessary to evaluate all guards of all transitions originating from that location. The guard evaluation determines which ports are enabled. Each transition will then provide a jump to a location block. At the coarse level, the elements that precede a block ℓ are the blocks that have transitions linking to ℓ, while the elements that succeed ℓ are all blocks to which ℓ is a predecessor. At the fine level, the elements that syntactically precede the transition are all transitions that lead to its block. The elements that succeed the transition are all transitions in the block to which it refers to. Given a set of transitions $M \subseteq B.trans$, we define:

- $\mathrm{origin}(M) = \{\tau.src \mid \tau \in M\}$ to be the set of the origin/source locations of M (blocks that contain the transitions in M);
- $\mathrm{dest}(M) = \{\tau.dest \mid \tau \in M\}$ to be the set of the destination locations of M (blocks to which the transitions lead);
- $\mathrm{siblings}(M) = \{\tau \in B.trans \mid \tau.src \in \mathrm{origin}(M)\}$ to be the set of transitions of B that have their source locations within the origin locations of M (selecting all transitions of the same block);
- $\mathrm{prev}(M) = \{\tau \in B.trans \mid \tau.dest \in \mathrm{origin}(M)\}$ to be the set of transitions of B that have their destination locations within the origin locations of M (transitions that lead to the block).

Example 11 (Syntactic representation). *Figure 4 shows an atomic component in the two views. We consider the set of transitions $M = \{t_3, t_5\}$ which is shown in bold. Figure 4a illustrates* $\mathrm{origin}(M)$, $\mathrm{dest}(M)$, $\mathrm{siblings}(M)$ *and* $\mathrm{prev}(M)$. *The origin set contains the locations from which the transitions in M are outbound: $\{\ell_0, \ell_2\}$. The destination set contains the locations to which the transitions in M lead to: $\{\ell_1, \ell_4\}$, and are highlighted with a pattern. The dotted transitions belong to* $\mathrm{siblings}(M)$, *they are all transitions originating from* $\mathrm{origin}(M)$ *i.e.,*



(a) portExecute(p_1) (b) portEnabled(p_1) (c) Combination

Figure 5: Matching Ports

t_0, t_2, t_3, t_4 *and* t_5. *They are the transitions in the same block as* M. *The dashed transitions belong to* $\mathrm{prev}(M)$, *they are* t_0 *and* t_4. *These transitions lead to* $\mathrm{origin}(M)$.

Definition 20 (Local pointcut selection). *A local pointcut expression selects a subset of* $B_k.trans$: $\mathrm{select}_\ell(B_k, lpc) =$ *match* lpc *with*

$$
\begin{array}{ll}
\mid \mathrm{atLocation}(\ell) & \to \{\tau \in B_k.trans \mid \tau.src = \ell\} \\
\mid \mathrm{readVarGuard}(x) & \to \mathrm{siblings}(\{\tau \in B_k.trans \mid x \in \mathrm{readguard}(\tau)\}) \\
\mid \mathrm{readVarFunc}(x) & \to \{\tau \in B_k.trans \mid x \in \mathrm{readvar}(\tau)\} \\
\mid \mathrm{write}(x) & \to \{\tau \in B_k.trans \mid x \in \mathrm{writevar}(\tau)\} \\
\mid \mathrm{portExecute}(p) & \to \{\tau \in B_k.trans \mid \tau.port = p\} \\
\mid \mathrm{portEnabled}(p) & \to \mathrm{siblings}(\{\tau \in B_k.trans \mid \tau.port = p\}) \\
\mid \phi \text{ and } \phi' & \to \mathrm{select}_\ell(B_k, \phi) \cap \mathrm{select}_\ell(B_k, \phi')
\end{array}
$$

Predicate $\mathrm{atLocation}(\ell)$ matches the transitions in block ℓ. Guards are evaluated at the level of the block. Therefore, to match $\mathrm{readVarGuard}(x)$, we first select the transitions that contain x in their guard, and then select their blocks with $\mathrm{siblings}()$. An update function, however, is at the level of transitions. Henceforth, $\mathrm{readVarFunc}(x)$ and $\mathrm{write}(x)$ select only the transitions which update function reads x and modifies x, respectively. For ports, the execution of a port is at the level of a transition, therefore $\mathrm{portExecute}(p)$ selects all transitions that contain the port p. However, the enablement of a port happens at the level of the block, since multiple ports can be enabled but only one executes, therefore $\mathrm{portEnabled}(p)$ extends the selection to the block of the transitions that would normally be selected by $\mathrm{portExecute}(p)$. When combining matches, the result must select transitions that are affected by both pointcuts. For this, the transitions from both matches are intersected to ensure that the result has transitions present in both. Note that since $\cap$ is associative and commutative, the set of obtained transitions is insensitive to the match order.

Example 12 (Matching ports). *Figure 5a shows (in red) the transitions matched with the pointcut* $\mathrm{portExecute}(p_1)$. *To match the execution of port* p_1, *all transitions labeled with* p_1 *are selected. These transitions are executed only if port* p_1 *is executed. Figure 5b shows the transitions*

matched with $\mathrm{portEnabled}(p_1)$. *Note that more transitions are selected since the enablement of* p_1 *may be followed by the execution of a port different than* p_1 *(e.g.,* p_2*). Port* p_1 *is enabled at* ℓ_2 *iff* g_2 *evaluates to* `true`*. Determining if* p_1 *is enabled requires pre-evaluating* g_2*. Additionally,* p_2 *may be executed while* p_1 *is enabled at* ℓ_2*. Then, the component may execute either* t_3 *or* t_4*. Therefore, the joinpoint must include* t_3 *and* t_4*. However, if* g_2 *evaluates to* `false`*,* p_1 *is not enabled. Thus the joinpoint must not include* t_3 *and* t_4*. To handle the evaluation of the guards at runtime and executing the advice properly, additional elements need to be instrumented, they are detailed in Section 6.4. Figure 5c shows the combination of the matches from the two, effectively showing* $\mathrm{portExecute}(p_1)$ *since the execution implies enablement.*

Proposition 3. $e \vDash lpc$ *iff* $e.\tau \in \mathrm{select}_\ell(B_k, lpc)$ *where* e *is a local event, and we assume that* lpc *does not contain* $\mathrm{portEnabled}(p)$.

This proposition states that a local event e is a local joinpoint ($e \vDash lpc$) iff its transition $e.\tau$ is syntactically selected (i.e. $\tau \in \mathrm{select}_\ell(B_k, lpc)$). Since state information is accessible only at runtime, determining enabled ports ($\mathrm{portEnabled}(p)$) requires additional instrumentation.

6.3. Local Advice

The local advice defines the possible actions to be injected at a local joinpoint. Similarly to a global advice (Definition 15), a local advice executes two functions before and after the local joinpoint. Moreover, we introduce a constraint to restrict the variables accessible to the advice functions. Advice variables consists of the variables of the atomic component and an extra set of inter-type variables V. Furthermore, in order to increase the expressiveness of the local advice and the refinement of local behavior, a local advice may change the location of the atomic component depending on a specific guard. However to ensure consistency, the location change happens once both functions of the advice have finished executing.

Definition 21 (Local advice). *A local advice* $ladv(B_k, V)$ *is a triple* $\langle F_b, F_a, R\rangle$*. It has access to* $X_{adv} = B_k.vars \cup V$*. It consists of:*

1. *a before function* F_b *such that* $(\mathrm{readvar}(F_b) \cup \mathrm{writevar}(F_b)) \subseteq X_{adv}$*;*
2. *an after function* F_a *such that* $(\mathrm{readvar}(F_a) \cup \mathrm{writevar}(F_b)) \subseteq X_{adv}$*;*
3. *and a set* R *of reset locations, defined as a set of tuples* $\langle \ell, g\rangle$ *where each tuple indicates that, after the end of the joinpoint, the component has to move to location* ℓ *if guard* g *evaluates to* `true`*.*

A local aspect binds a local pointcut to a local advice and defines the inter-type variables.

Definition 22 (Local aspect). *A local aspect is a tuple* $\langle B_k, lpc, V, ladv(B_k, V)\rangle$ *where* B_k *is an atomic component,* lpc *is a pointcut expression,* V *is the set of inter-type variables,* $ladv(B_k, V)$ *is the local advice to apply on the joinpoints.*

Example 13 (Local aspect). *Let us consider the local aspect:*

$$\langle B_k, \mathrm{atLocation}(\ell_2), \{v_0\}, \langle F_b, F_a, \{\langle \ell_1, x > 1\rangle\}\rangle\rangle$$

It applies to component B_k*. The local pointcut* $\mathrm{atLocation}(\ell_2)$ *selects any joinpoint where* B_k *is at location* ℓ_2*. The advice* $\langle F_b, F_a, \{\langle \ell_1, x > 1\rangle\}\rangle$ *specifies that i) right before* B_k *enters* ℓ_2*,* F_b *must execute, and ii) after* B_k *exits* ℓ_2*,* F_a *must execute, and then iii) if* $x > 1$ *holds at runtime the component must move to location* ℓ_1*.*

6.4. Local Weaving

Similarly to global weaving (discussed in Section 5.3), the local weaving procedure instruments a BIP system around an atomic component. It therefore weaves a local advice to local joinpoints selected by local pointcut expressions.

Edit frame.. We recall that a local advice (see Definition 21) defines extra update functions to execute, and the possibility to change the location of the atomic component. In order to match local joinpoints with local pointcuts, we need to instrument the atomic component while using the transition as the unit (Definition 20). Then, we need to define where the instrumentation occurs relative to the transition. Using the syntactic representation in Section 6.2, we define *edit points*. Edit points provides hints to the weaving procedure to determine which elements need to be modified, so that during runtime, the advice executes appropriately. We identify four edit points:

- RUN (Runtime) specifies that instrumentation is to detect the state at runtime, and it applies at the block-level of the match (i.e., the location block of each transition in the match);
- PE (Previous End) specifies that the instrumentation applies at the level of the transitions that lead to the location blocks that include the match and specifically at the end of their update function;
- CB (Current Begin) specifies that the instrumentation applies at the level of the transitions in the match, at the beginning of their update function;
- CE (Current End) specifies that the instrumentation applies similarly to CB but instead, at the end of the update function.

The set of edit points is $EP = \{PE, CB, CE, RUN\}$. The edit point RUN is a special hint that is associated with runtime information. We use it to determine if a port is enabled as it requires the evaluation of the guards during runtime. We use it to encode an "if else" statement, so that the guards are evaluated in a temporary state. The temporary state is needed as advices have an update function for "before", and since we cannot wait for the evaluation to happen to execute the advice, we have to pre-evaluate the guards to determine if a port is enabled.

The *edit frame* indicates the start and end edit points, it informally determines the start and end of the region necessary to match the pointcut, so that we can weave F_b and F_a respectively. An edit frame is determined depending on the local pointcut expression as follows:

Definition 23 (Edit frame). *The edit frame is a pair of edit points $\langle e_1, e_2\rangle$ corresponding to the update functions of the local advice: F_b and F_a respectively. It is defined as:* $\mathrm{edit}(lpc) =$ *match lpc with:*

$$
\begin{array}{lll}
\mid \mathrm{atLocation}(\ell) & \to \langle PE, CB\rangle & \mid \mathrm{readVarFunc}(x) \quad \to \langle CB, CE\rangle \\
\mid \mathrm{readVarGuard}(x) & \to \langle PE, CB\rangle & \mid \mathrm{write}(x) \quad \to \langle CB, CE\rangle \\
\mid \mathrm{portEnabled}(p) & \to \langle RUN, CB\rangle & \mid \mathrm{portExecute}(p) \quad \to \langle CB, CE\rangle \\
\mid \phi \text{ and } \phi' & \to \langle \max(e_1, e_1'), \max(e_2, e_2')\rangle &
\end{array}
$$

where $\langle e_1, e_2\rangle = \mathrm{edit}(\phi)$ and $\langle e_1', e_2'\rangle = \mathrm{edit}(\phi')$, with the strict ordering $PE \prec CB \prec CE \prec RUN$.

An extra update function in the case of a location ($\mathrm{atLocation}(\ell)$) happens at the end of any transition that leads to the block ℓ. This ensures that it executes prior to the block ℓ. We consider the end of the block whenever a transition is to execute, i.e., at the beginning of its update function. This is done similarly for guards ($\mathrm{readVarGuard}(x)$) as guards are evaluated at the block level. In the case of an update function F, any extra update function applies before F at its beginning, and after F at its end. Therefore, for a variable read ($\mathrm{readVarFunc}(x)$) or modification ($\mathrm{write}(x)$) any extra update function should happen before F. We note that this is the finest level of granularity, we do not inspect the sequence of assignments in the update function, only its beginning and end. Similarly, a port execution ($\mathrm{portExecute}(p)$) is the execution of an update function of a transition so the edit points are identical. Since state information is accessible only at runtime, determining enabled ports ($\mathrm{portEnabled}(p)$) requires modification at the block level, mostly by inserting additional blocks that pre-evaluate the guards and do some extra computation. Thus, we associate the edit frame $\langle RUN, CB\rangle$ to $\mathrm{portEnabled}$ so that guards are pre-evaluated appropriately.

In the case of a combination of pointcuts, we must ensure that the frame can capture *both pointcuts*. Thus, we define a strict order on EP: $PE \prec CB \prec CE \prec RUN$. The purpose of the order is to select the most relevant elements for weaving, considering our representation of the syntax of BIP (described in Section 6.2). It can be seen as the scope of elements that need to be instrumented. Since RUN requires runtime checks, it always requires the most modification, therefore it is the maximal element. At the level of transitions, the order follows from the precedence: the start of an update function (CB) precedes its end (CE), and the entire transition is preceded by another transition that leads to its block (PE). At the level of the block, the transitions that lead to the block, precede it (PE). A combination of two frames must start when the elements overlap (i.e., the most delayed edit frame), and end once both end (i.e., at the most delayed one). Also, the combination must include all elements that needs to be instrumented to match both frames. To fulfill both of these conditions, we use $\max$, as it will include the hint to the maximal elements needed to be instrumented. RUN includes the instrumentation necessary at the entire block level for detecting $\mathrm{portEnabled}$ at runtime. Therefore, combining RUN with any pointcut frame will still require runtime information. Therefore, RUN is defined as the maximum. Combining port enabled with the other frames adds the port enablement condition on the existing condition (as per the semantics of $\mathrm{portEnabled}$ in Definition 19), therefore to capture the two cases: $\langle PB, CB\rangle$ and $\langle CB, CE\rangle$, we discriminate using the second point. When used alone, we consider $\mathrm{portEnabled}$ to end similarly to $\mathrm{atLocation}$ and $\mathrm{readVarGuard}$, at the start of transition execution.

Note that, since $\max$ is associative and commutative the order does not matter. Exhausting all possible combinations with $\max$, only the following frames are possible: $\langle PE, CB\rangle$, $\langle CB, CE\rangle$, $\langle RUN, CB\rangle$, and $\langle RUN, CE\rangle$.

Example 14 (Edit frames). *We consider the pointcut expression:* $\mathrm{atLocation}(\ell_1)$ *and* $\mathrm{write}(x)$. *These expressions have the frames* $\langle PE, CB\rangle$ *and* $\langle CB, CE\rangle$*, respectively. This means that* $\mathrm{atLocation}(\ell_1)$ *starts right after the previous transitions leading to* ℓ_1 *have finished executing their update function, while for* $\mathrm{write}(x)$ *it starts right after the transition executes (i.e., CB, at the start of its update function). Their combination must start when they both start, in this case* CB*. If we consider the start to be the earliest, then we can still pass* PE *but it is possible to execute another transition that does not match* $\mathrm{write}(x)$*. This is not consistent with the start of* $\mathrm{write}(x)$*. They both end after (1)* $\mathrm{atLocation}(\ell_1)$ *is over and (2)* $\mathrm{write}(x)$ *is over, therefore they end when the transition ends (*CE*).*

Overview.. Recall from Definition 21 that both of the update functions F_b and F_a of an advice have read/write access to a component variables. It is also possible that F_b and F_a belong to different transitions. Therefore, it could be possible to update the variables in F_b in a way that ensures F_a is never executed. The first requirement of the weaving procedure is to ensure that F_a will always execute once F_b has executed.

Remark 3 (Deadlocks). *It is ensured that function F_a executes after F_b if the user's advice does not result in a deadlock state. For example, the before computation may modify the state so as to have no outgoing enabled transition. Hence, the after computation is not executed. Since weaving transforms a BIP model, it is possible to use verification tools (such as DFinder [7]) on the transformed model to check for deadlock freedom.*

In order to apply the requirement, the general weaving strategy uses one boolean variable per aspect b_{aop}. The variable b_{aop} is set to `false` right before reaching the joinpoint and set to `true` upon joinpoint entry, indicating a pointcut match. It is then up to the weaving procedure to unset if necessary. We use for brevity: $F_{\text{set}} = \langle b_{\text{aop}} := \texttt{true} \rangle$ and $F_{\text{clear}} = \langle b_{\text{aop}} := \texttt{false} \rangle$. In addition to b_{aop} we define the extra port ip. The ip port is associated with a singleton interaction with the highest priority in the system. It is used to create high priority transitions in the component creating deterministic behavior.

In the following, we begin by weaving the reset location pairs, since it is independent of execution frames. Later, we consider the weaving of the advice $adv = \langle F_b, F_a, \{\} \rangle$ on a set of transitions M for each of the four pairs of frames: $\langle PE, CB \rangle$, $\langle CB, CE \rangle$, $\langle RUN, CB \rangle$, $\langle RUN, CE \rangle$. We fix T' and L' to be the set of transitions and locations of the new atomic component respectively. A set of locations L_{R} contains the locations (blocks) on which the pointcut ends, so that it is possible to weave the reset locations. An injective function $\text{m} : B_k.trans \to 2^{T'}$ relates $B_k.trans$ to the new transitions by transforming them, creating extra transitions or copying them.

Weaving reset location pairs.. We consider a set R of reset location pairs, and a set of locations L_{R}. The reset location transitions are defined as follows:

$$\text{wReset}(R, L_{\text{R}}) = \bigcup_{\langle guard, dest \rangle \in R} (\{\langle \ell, ip, b_{\text{aop}} \wedge guard, F_{\text{clear}}, dest \rangle \mid \ell \in L_{\text{R}}\}).$$

The transitions are guarded by b_{aop} so as to execute only if the pointcut matched. They execute on ip so as to have priority over other transitions at the location. Once executed they invoke F_{clear} to indicate that the pointcut has ended. Consequently it avoids a deadlock when the location resets to itself.

Example 15 (Weaving reset location). *Figure 6 shows the weaving of reset locations, namely* $\text{wReset}(\{\langle \ell_4, g_a \rangle, \langle \ell_1, g_b \rangle\}, \{\ell_1, \ell_2\})$. *In total, four transitions are created as we have two locations and two reset pairs. Transitions are highlighted differently for each destination location.*

We now elaborate on the weaving of each edit frame by detailing the transitions that are modified, the extra locations created and specifying which locations are selected to apply the reset location transformation on.

Weaving $\langle CB, CE \rangle$.. The edit frame $\langle CB, CE \rangle$ applies at the level of transitions in M. Therefore, F_b (resp. F_a) is simply added at the beginning (resp. end) of F, resulting in $F' = \langle F_b, F, F_a \rangle$. All transitions leading to the transitions in M, namely $\text{prev}(M)$ must invoke F_{clear}

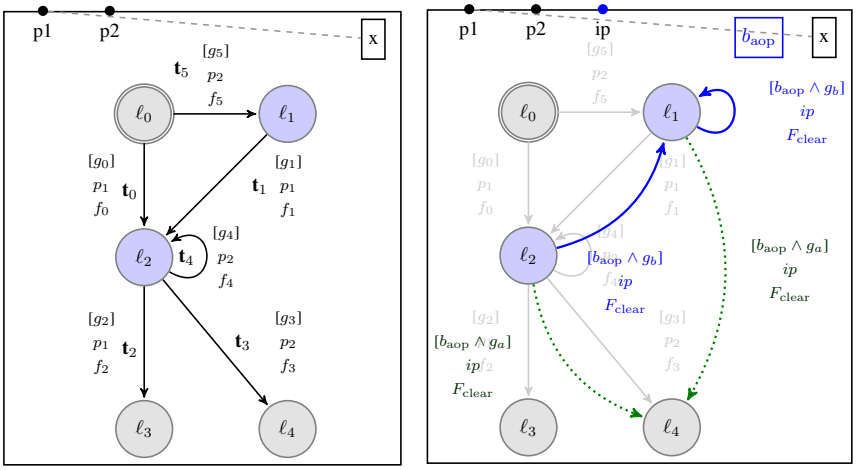


Figure 6: Weaving Reset Locations

after finishing their computation. Care should be taken in the case of loops as they are both in the match and lead to the match. F_{set} is executed to indicate that the joinpoint matched followed by F_a. No additional locations are necessary, $L' = B_k.locs$. L_{R} consists of the locations to which any transition in M leads, they are then $\text{dest}(M)$. We define m as follows:

$$m(t = \langle \ell, p, g, F, \ell' \rangle) = \begin{cases} \{\langle \ell, p, g, FF_{\text{clear}}, \ell' \rangle\} & \text{if } t \in \text{prev}(M) \setminus Mm \\ \{\langle \ell, p, g, F_b F F_{\text{set}} F_a, \ell' \rangle\} & \text{if } t \in M, \\ \{t\} & \text{otherwise.} \end{cases}$$

The new set of transitions is then $T' = \bigcup_{t \in B_k.trans} m(t)$. The whole procedure is defined as: $\text{wframe}_{\text{cur}}(B_k, M, F_b, F_a) = \langle T', L', L_{\text{R}}, m \rangle$.

Weaving $\langle PE, CB \rangle$.. Both PE and CB apply at the level of transitions. PE indicates that F_b must be woven on the transitions leading to those in M namely $\text{prev}(M)$, at the end of their update function. CB indicates that F_a must be woven on the transitions in M, at the beginning of their update function. In this case, loops require more instrumentation. Let the set of loop transitions be $T_{\text{L}} = \text{prev}(M) \cap M$, if we have such transitions $T_{\text{L}} \neq \emptyset$, we create the set of extra locations $L_{\text{temp}} = \{\ell^{\perp} \mid \ell \in \text{origin}(T_{\text{L}})\}$. For a given loop transition $\langle \ell, p, g, F, \ell \rangle$, we set the update function to $\langle F_a, F_{\text{set}}, F \rangle$ and change the destination to $\ell^{\perp}$ and add a transition that executes F_b on port ip from $\ell^{\perp}$ to ℓ. Thus if a reset location is to happen, it would happen on $\ell^{\perp}$, without executing function F_b again.

$$\begin{aligned} L_{\text{R}} &= \begin{cases} \text{dest}(M) \setminus \text{origin}(T_{\text{L}}) \cup L_{\text{temp}} & \text{if } T_{\text{L}} \neq \emptyset \\ \text{dest}(M) & \text{otherwise} \end{cases} \\ L' &= \begin{cases} B_k.locs \cup L_{\text{temp}} & \text{if } T_{\text{L}} \neq \emptyset \\ B_k.locs & \text{otherwise} \end{cases} \end{aligned}$$

$$m(t = \langle \ell, p, g, F, \ell' \rangle) = \begin{cases} \{\langle \ell, p, g, FF_{\text{clear}}F_b, \ell' \rangle\} & \text{if } t \in \text{prev}(M) \setminus M \\ \{\langle \ell, p, g, F_aF_{\text{set}}F, \ell^{\perp} \rangle\} & \text{if } t \in M \cap \text{prev}(M) \\ \{\langle \ell, p, g, F_aF_{\text{set}}F, \ell' \rangle\} & \text{if } t \in M \setminus \text{prev}(M) \\ \{t\} & \text{otherwise} \end{cases}$$

$$T' = \{m(t) \mid t \in B_k.trans\} \cup \{\langle \ell^{\perp}, ip, \texttt{true}, F_{\text{clear}}F_b, \ell \rangle \mid \ell^{\perp} \in L_{\text{temp}}\}$$

The whole procedure is defined as: $\text{wframe}_{\text{prev}}(B_k, M, F_b, F_a) = \langle T', L', L_{\text{R}}, m \rangle$.

Weaving $\langle RUN, CB \rangle$.. This frame requires block instrumentation to handle portEnabled. RUN indicates that additional computations needs to be handled. These computations refer to port enablement since only matching at least one port enablement can lead to RUN. In the first step, we define the selected ports. Selected ports are the ports matched as part of the pointcut expression lpc using $\text{sp}(lpc) =$ match lpc with

$\|\text{atLocation}(\ell)$	$\to \emptyset$	$\|\text{readVarFunc}(x)$	$\to \emptyset$	$\|\text{readVarGuard}(x)$	$\to \emptyset$
$\|\text{write}(x)$	$\to \emptyset$	$\|\text{portEnabled}(p)$	$\to \{p\}$	$\|\text{portExecute}(p)$	$\to \emptyset$
$\|\phi$ and ϕ'	$\to \text{sp}(\phi)$	$\cup \text{sp}(\phi')$			

First we begin by creating the guard expression for the enabled ports. A port p is enabled in a location ℓ if there exists at least one transition which guard evaluates to `true`. In the case of multiple ports, they must all be enabled.

$$\text{mkGuard}(SP, \ell, M) = \bigwedge_{p \in SP} \Big(\bigvee_{\tau \in M \wedge \tau.src = \ell \wedge \tau.port = p} (\tau.guard) \Big)$$

Second, since port enablement can be detected only when the guards are evaluated at the location, to execute function F_b we need to pre-evaluate the guards before entering the location. To match a port enablement at a location ℓ, we create a temporary location $\ell^{\perp}$ that pre-evaluates the guard. By applying this to the entire match M, we have $L_{\text{temp}} = \{\ell^{\perp} \mid \ell \in \text{origin}(M)\}$. We then connect each $\ell^{\perp}$ to ℓ with two transitions, the first executes F_b and does F_{set}, indicating the pointcut match, and the second does F_{clear}. Both these transitions execute on ip so as to not be enabled or execute any existing port. The added transitions are:

$$T_{\text{cr}} = \bigcup_{\ell^{\perp}} (\{\langle \ell^{\perp}, ip, g, F_{\text{set}}F_b, \ell \rangle, \langle \ell^{\perp}, ip, \neg g, F_{\text{clear}}, \ell \rangle\} \text{ with } g = \text{mkGuard}(\text{sp}(lpc), \ell, M))$$

Third, since port enablement is determined dynamically, we simulate an `if/else` construct. Outgoing transitions from ℓ are duplicated. This results in two versions. The first checks for the joinpoint match (if b_{aop} holds) and applies F_a. The second checks for ($b_{\text{aop}} =$ `false`) and does not apply F_a weaving in CB.

Lastly, all the incoming transitions from ℓ are redirected from ℓ to $\ell^{\perp}$ so as to reach $\ell^{\perp}$ to pre-evaluate the guard before ℓ. Thus, we get the resulting new set of locations L' wand transitions T'.

$$L' = B_k.locs \cup L_{\text{temp}} \qquad T' = \{m(t) \mid t \in B_k.trans\} \cup T_{\text{cr}}$$

with $t = \langle \ell, p, g, F, \ell' \rangle$ and

$$m(t) = \begin{cases} \{\langle \ell, p, b_{\text{aop}} \wedge g, F_aF, \ell' \rangle, \langle \ell, p, \neg b_{\text{aop}} \wedge g, F, \ell' \rangle\} & \text{if } t \in M \\ \{\langle \ell, p, g, F, \ell'^{\perp} \rangle\} & \text{if } t \in \text{prev}(M) \setminus M \\ \{t\} & \text{otherwise} \end{cases}$$

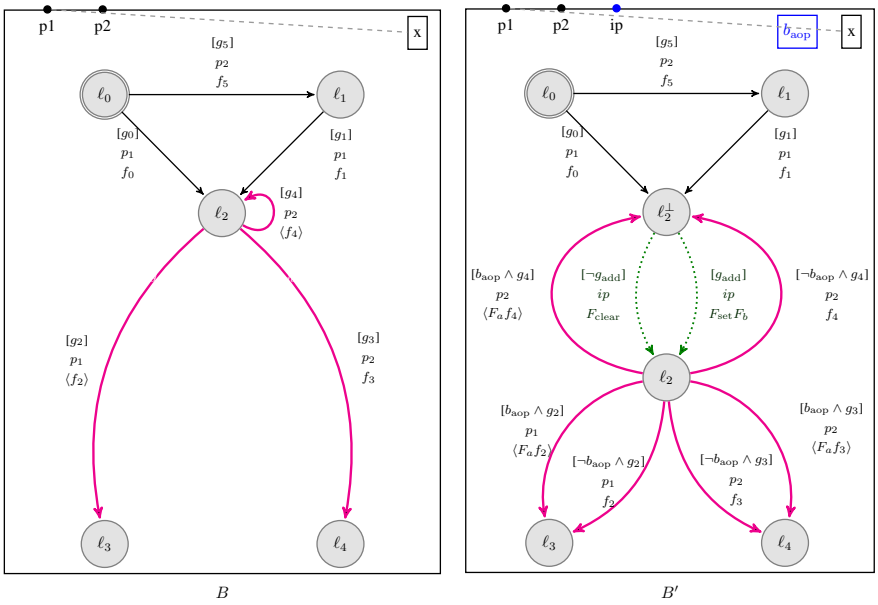


Figure 7: Weaving $\langle RUN, CB \rangle$

In the case of loops, a reset location should happen on the temporary locations.

$$L_{\text{R}} = \text{dest}(M) \setminus \text{origin}(M) \cup L_{\text{temp}}$$

The whole procedure is defined as: $\text{wframe}_{\text{runb}}(B_k, M, F_b, F_a, lpc) = \langle T', L', L_{\text{r}}, m \rangle$.

Example 16 (Dynamic weave). *Figure 7 depicts the weaving of the advice* $\langle F_b, F_a, \emptyset \rangle$ *with the pointcut* $lpc = \text{atLocation}(\ell_2)$ and $\text{portEnabled}(p_1)$ and $\text{portEnabled}(p_2)$. *The selected ports are* $\text{sp}(lpc) = \{p_1, p_2\}$, *the origin is determined,* $\text{origin}(\{t_2, t_3, t_4\}) = \{\ell_2\}$. *The condition for the enabled ports is:* $g_{\text{add}} = \text{mkGuard}(\{p_1, p_2\}, \ell_2, \{t_2, t_3, t_4\}) = g_2 \wedge (g_3 \vee g_4)$. *Both ports* $\{p_1, p_2\}$ *are enabled when* p_1 *is enabled (*g_2 *is* `true`*) and* p_2 *is enabled (*$g_3 \vee g_4$ *is* `true`*). We create a set of temporary locations:* $L_{\text{temp}} = \{\ell_2^\perp\}$. *Two transitions are created per location. The first executes iff the ports are both enabled (pointcut matched), its update function is* F_b *and* F_{set}. *The second executes iff one of the ports is not enabled (pointcut not matched), its update function does* F_{clear}. *Transitions with* ℓ_2 *as destination are redirected to* $\ell_2^\perp$. *Then, we copy over the originally outgoing transitions from* ℓ_2, *creating two versions of them: one has* F_a *and is guarded by* b_{aop} *and another executes normally and is guarded by* $\neg b_{\text{aop}}$.

Weaving $\langle RUN, CE \rangle$.. The frame is woven similarly to $\langle RUN, CB \rangle$. The RUN edit point indicates that portEnabled must hold, so the transformations described in the previous part are similar. However, we modify the order of execution of F_b and F_a. Since F_b executes on CB, we do not execute it on the added transitions that detect port enabled.

$$T_{\text{cr}} = \bigcup_{\ell^\perp} (\{\langle \ell^\perp, ip, g, F_{\text{set}}, \ell \rangle, \langle \ell^\perp, ip, \neg g, F_{\text{clear}}, \ell \rangle\} \text{ with } g = \text{mkGuard}(\text{sp}(lpc), \ell, M)).$$

Instead, F_b needs to be added to the transition which matches the port enablement. Therefore, F_b and F_a are added to the duplicated transitions that match the guard g.

$$L' = B_k.locs \cup L_{\text{temp}} \qquad T' = \{m(t) \mid t \in B_k.trans\} \cup T_{\text{cr}}$$

with $t = \langle \ell, p, g, F, \ell' \rangle$ and

$$m(t) = \begin{cases} \{\langle \ell, p, b_{\text{aop}} \wedge g, F_b F F_a, \ell' \rangle, \langle \ell, p, \neg b_{\text{aop}} \wedge g, F, \ell' \rangle\} & \text{if } t \in M, \\ \{\langle \ell, p, g, F, \ell'^{\perp} \rangle\} & \text{if } t \in \text{prev}(M) \setminus M, \\ \{t\} & \text{otherwise.} \end{cases}$$

When considering reset location, loops and temporary locations must be adjusted similarily to weaving $\langle RUN, CB \rangle$:

$$L_{\text{R}} = \text{dest}(M) \setminus \text{origin}(M) \cup L_{\text{temp}}$$

The whole procedure is defined as: $\text{wframe}_{\text{runa}}(B_k, M, F_b, F_a, lpc) = \langle T', L', L_{\text{r}}, m \rangle$.

Weaving a local aspect.. The local weave operation weaves an advice on set of transitions M with the set of intertype variables V and a local pointcut expression lpc.

Definition 24 (Local Weave). *The local weave is defined as:*

$$\langle \mathcal{C}', \text{m} \rangle = \text{weave}_{\ell}(\mathcal{C}, B_k, V, lpc, M, ladv(B_k, V) = \langle F_b, F_a, R \rangle)$$

where $\mathcal{C}' = \pi(\gamma'((\mathcal{B} \setminus B_k) \cup B'_k))$ is the resulting composite component; with:

- $B'_k = \langle P \cup \{ip\}, L', X \cup V \cup \{b_{\text{aop}}\}, T' \cup T_{\text{R}} \rangle$ *is the resulting atomic component;*
- b_{aop} *is the aspect boolean variable described earlier;*
- $\langle T', L', L_{\text{R}}, m \rangle = \begin{cases} \text{wframe}_{\text{cur}}(B_k, M, F_b, F_a) & \textit{if } \text{edit}(lpc) = \langle CB, CE \rangle \\ \text{wframe}_{\text{prev}}(B_k, M, F_b, F_a) & \textit{if } \text{edit}(lpc) = \langle PE, CB \rangle \\ \text{wframe}_{\text{runb}}(B_k, M, F_b, F_a, lpc) & \textit{if } \text{edit}(lpc) = \langle RUN, CB \rangle \\ \text{wframe}_{\text{runa}}(B_k, M, F_b, F_a, lpc) & \textit{if } \text{edit}(lpc) = \langle RUN, CE \rangle \end{cases}$
- $T_{\text{R}} = \text{wReset}(R, L_{\text{R}})$ *is obtained from weaving reset locations;*
- $\gamma' = \gamma \cup \{a_{ip}\}$ *where* $a_{ip} = \langle ip, \texttt{true}, \langle\rangle \rangle$ *is the high priority interaction for the aop port;*
- $\pi' = \pi \cup \{\langle a, a_{ip} \rangle \mid a \in \pi\}$ *is the new set of priorities.*

Weaving a local aspect on a composite component $\mathcal{C} = \pi(\gamma(\mathcal{B}))$ is defined as:

$$\mathcal{C}' = \mathcal{C} \triangleleft_{\ell} \langle B_k \in \mathcal{B}, lpc, V, ladv(B_k, V) \rangle$$

where: $\mathcal{C}'$ is the result from the local weave: $\langle \mathcal{C}', \text{m} \rangle = \text{weave}_{\ell}(\mathcal{C}, B_k, V, lpc, \text{select}_{\ell}(B_k, lpc), ladv(B_k, V))$.

The new atomic component B'_k has one extra port (ip), and has $B_k.vars \cup V \cup \{b_{\text{aop}}\}$ as the set of variables. The edit frame is determined using operator edit (Definition 23) and transitions and locations are instrumented accordingly. The obtained composite component $\mathcal{C}'$ has one extra singleton interaction a_{ip} associated with port ip. Additionally, interaction a_{ip} is given the highest priority w.r.t. predefined interactions.

Correctness of local weave.. Informally, to verify the correct addition of the advice w.r.t joinpoint matching, we need to first verify that the before and after update functions were placed correctly when a joinpoint is matched during runtime. This is similar to the correctness of global weaving expressed in Proposition 2. However, we note that the before update function (F_b) must, in some cases, execute in the local event preceding the match. Secondly, in the presence of reset location, the local event succeeding the matched event should indicate that the local component is in the appropriate location. As such, the order of the events is also checked in the case of local weaving.

We first need to make the distinction between the frames that weave F_b on the previous transitions with those that do not.

We define predicate $\mathrm{early}(lpc) = \begin{cases} \texttt{false} & \text{if } \mathrm{edit}(lpc) \in \{\langle CB, CE\rangle, \langle RUN, CE\rangle\}, \\ \texttt{true} & \text{otherwise.} \end{cases}$

We consider $\mathcal{E}'_k$ (resp. $\mathcal{E}_k$) to be reachable events in B'_k (resp. B_k). We define $\mathrm{rem}_k : \mathcal{E}'_k \times \mathcal{F} \times \mathcal{F} \to \mathcal{E}_k \cup \{\epsilon\}$, the function that removes the local advice from a local event.

$$\mathrm{rem}_k(\langle\langle l, v\rangle, \tau, q\rangle, F_b, F_a) = \begin{cases} \langle\langle l, v'\rangle, \tau', q'\rangle & \text{if } l \in B_k.locs, \\ \epsilon & \text{otherwise.} \end{cases}$$

The constructed event is the following:

- v' excludes the valuations of the inter-type V from v while the location is maintained.
- If the event has a location not found in $B_k.locs$, then a similar event cannot be constructed. The resulting event is ϵ.
- $\tau' = \langle l, \tau.port, g, F, l'\rangle$ is the similar transition where g does not contain b_{aop}, F does not contain $F_{\mathrm{set}}, F_{\mathrm{clear}}, F_b, F_a$ and l' is any location (since q' is never matched against a joinpoint so it is not relevant).

By removing the advice, we ensure that lpc must not match read/writes introduced by F_b and F_a.

We use $\mathrm{before}(e_i, F, d)$ and $\mathrm{after}(e_i, F, d)$ where e_i is a local event, F is the before or after update function, and d is the value of the predicate early, to verify the correct application of the before and after update functions respectively.

$$\begin{aligned} \mathrm{before}(e_i, F, d) \text{ iff } & (b \wedge i \neq 0 \wedge \exists F' : e_{i-1}.\tau.\mathit{func} = \langle F', F\rangle) \\ & \vee (\neg b \wedge \exists F' : e_i.\tau.\mathit{func} = \langle F, F'\rangle) \\ \mathrm{after}(e_i, F, d) \text{ iff } & (b \wedge \exists F' : e.\tau.\mathit{func} = \langle F, F'\rangle) \vee (\neg b \wedge \exists F' : e.\tau.\mathit{func} = \langle F', F\rangle) \end{aligned}$$

In the case where the predicate $\mathrm{early}(lpc)$ holds, the predicate $\mathrm{before}(e_i, F, d)$ checks if the function of the preceding event's update function ($e_{i-1}.\tau.\mathit{func}$) ends with F, with the exception of the first event ($i \neq 0$), while the predicate after checks that the current event's function ($e_i.\tau.\mathit{func}$) starts with F. In the case where the predicate $\mathrm{early}(lpc)$ does not hold, the predicate before (resp. after) ensures that the current event's update function ($e_i.\tau.\mathit{func}$) starts (resp. ends) with F.

For the case of a reset location pair $r = \langle \mathit{guard}, \mathit{loc}\rangle$ we check if its guard holds `true`on the next event. If it does, we verify that the location $e_{i+1}.l'$ is the destination location in a reset location pair.

$$\mathrm{reset}(e_i, r) \text{ iff } (\mathit{guard}(e_{i+1}.v) \implies e_{i+1}.l' = \mathit{loc})$$

We can now express the correct application as follows:

Proposition 4. *Correctness of local advice weaving. Consider T' to be the sequence of global events of the BIP System $\langle \mathcal{C}', Q_0 \rangle$ where $\mathcal{C}' = \mathcal{C} \triangleleft_\ell \langle B_k, lpc, V, ladv(B_k, V) \rangle$ and B'_k is the new atomic component with F_b and F_a as the advice before and after update functions. Let $T'_k = \text{map}(T', B'_k) = (e_0 \cdot e_1 \cdot \ldots)$, $d = \text{early}(lpc)$, we have $\forall e_i \in T'_k$, $e'_i = \text{rem}_k(e'_i, F_b, F_a)$:*

$$(e'_i \neq \epsilon \wedge e'_i \vDash lpc) \textit{ iff } \text{before}(e_i, F_b, d) \wedge \text{after}(e_i, F_a, d) \tag{1}$$

$$(e'_i \neq \epsilon \wedge e'_i \vDash lpc) \implies (R \neq \emptyset \implies \exists r \in R : \text{reset}(e_i, r)) \tag{2}$$

We begin by constructing an event $e'_i = \text{rem}_k(e_i, F_b, F_a)$ from e_i by removing the local advice. The proposition states that e'_i is a joinpoint ($e_i \vDash lpc$) in the original system iff the update function associated with e_i verifies the rules for before ($\text{before}(e_i, F_b, d)$) and after ($\text{after}(e_i, F_a, d)$). In the presence of reset location pairs (2), if the event e'_i is a joinpoint in the original system, the next event includes at least one reset location pair ($R \neq \emptyset \implies \exists r \in R : \text{reset}(e_i, r)$).

7. Weaving Strategies

An aspect is the single association of a pointcut expression to a joinpoint. However, when weaving more than one aspect, specific problems and extra considerations arise. This section identifies possible issues when weaving multiple aspects and presents two basic procedures to coordinate the weaving.

Furthermore, this section can be seen as a preliminary method to provide a grouping of aspects and basic strategies for weaving them. It can be seen as the interface between the user and the transformations described earlier in the paper for the two views. As such, while we present a basic overview of the strategies to combine and weave multiple aspects, we note that it is possible to use the transformations to build more complex ones that best suit the users' needs.

7.1. Interference

Recall that multiple concerns may happen at one joinpoint. This can be seen as the *tangling* phenomenon. When a new concern is added to the joinpoint, it is possible to have existing concerns at the same joinpoint. This situation is referred to as *interference*. Since not all concerns are independent, interference is an important issue to study.

Example 17 (Interference). *Consider $gpc = \langle \{p_1\}, \emptyset, \emptyset \rangle$, $V = \{x\}$, the two global aspects $GA_1 = \langle \mathcal{C}, V, gpc, \langle F_b = \langle x := 3 \rangle, F_a \rangle \rangle$, and $GA_2 = \langle \mathcal{C}, V, gpc, \langle F'_b = \langle x := 2 \rangle, F'_a \rangle \rangle$. Both of these aspects' advices operate on the same inter-type variable x and on the same matched joinpoints. There are four possibilities for weaving the advices, depending on the order of F_b, F'_b, F_a and F'_a. In the case of F_b and F'_b if we have $\langle F_b, F'_b \rangle$ (resp. $\langle F'_b, F_b \rangle$) then x will be 2 (resp. 3) at runtime.*

Defining weaving strategies helps to deal with interference in a more predictable way. To do so, we examine in the following: (1) a modular unit that groups aspects, and (2) the operations that weave multiple aspects.

7.2. Containers

Aspect containers encapsulate a group of aspects. Local containers (resp. global containers) apply to local (resp. global) aspects. Aspect containers seek to group interfering aspects and define extra restrictions so as to manage their weaving. By doing so, we expose multiple aspects to the user as a coherent unit.

Definition 25 (Global and local containers). *A global container is a tuple* $\langle \langle GA_1, \ldots, GA_n \rangle, V' \rangle$ *such that any* $GA_j \in \{GA_1, \ldots, GA_n\}$ *has the inter-type* V'*. A local container is a tuple* $\langle \langle LA_1, \ldots, LA_m \rangle, B, V \rangle$ *such that any* $LA_i \in \{LA_1, \ldots, LA_m\}$ *is applied to an atomic component* B *and has the inter-type* V*.*

Containers define an order on the aspects they encapsulate. They permit the definition of a weaving order for aspects. Moreover, containers ensure that aspects share the same inter-type variables. Sharing allows the inter-type to be encapsulated in the container. In the case of local containers, local aspects are required to operate on the same atomic component encouraging encapsulation. The local aspects operating on different atomic components do not interfere and cannot share inter-type variables.

7.3. Weaving Procedures

The weaveSerial *procedure..* Given a global or local container, one example of weaving procedure weaves aspects in the order they are contained. The aspects are presented in a sequence $\langle asp_1, \ldots, asp_n \rangle$ where $asp_1, \ldots, asp_n$ are all either global or local aspects.

Definition 26 (weaveSerial**).** *Depending on the aspect type we have two operations:*

- *The procedure* weaveSerial_ℓ *applied to a sequence of local aspects* $\langle asp_1, \ldots, asp_n \rangle$ *is:* $\mathcal{C}' = \text{weaveSerial}_\ell(\mathcal{C}, \langle asp_1, \ldots, asp_n \rangle) = ((((\mathcal{C} \lhd_\ell asp_1) \lhd_\ell asp_2) \lhd_\ell ..) \lhd_\ell asp_n)$
- *The procedure* weaveSerial_g *applied to a sequence of global aspects* $\langle asp_1, \ldots, asp_m \rangle$ *is:* $\mathcal{C}' = \text{weaveSerial}_g(\mathcal{C}, \langle asp_1, \ldots, asp_m \rangle) = ((((\mathcal{C} \lhd_g asp_1) \lhd_g asp_2) \lhd_g ..) \lhd_g asp_m)$

The weaveSerial procedure allows the pointcut of any aspect asp_i ($i \in \{2, \ldots, n\}$) to match changes introduced by all aspects asp_j prior to it (for $j < i$).

While Definition 26 presents a simple weaving strategy, we can consider alternative strategies. Firstly, we can consider that the matching of the pointcut and the weaving are separate. This is the case for weaveAll presented in the next paragraph. Secondly, we note that it is possible to modify the order of the aspects in the sequence. For example, it is possible to order aspects by (1) the type of the pointcut's edit frame or (2) by whether or not their advice performs only read or also modifies the variables.

The weaveAll *procedure..* The weaving procedure for local aspects introduces new locations and transitions in the cases of reset locations and port enabled. The second compositional approach weaveAll allows the pointcut of a local aspect asp_i in the sequence to not match extra transitions added by asp_j (for $j : j < i$). While weaveSerial can be seen as iteratively doing match and weave for each aspect, weaveAll makes it is possible to match the joinpoints of the entire sequence of aspects before weaving them. The procedure weaveAll matches all the pointcuts of the sequence, then weaves the aspects according to their order based on their original match projected onto the new component. For a local aspect $LA = \langle B, pc, V, F_b, F_a, R \rangle$ we denote $B, pc, V, \langle F_b, F_a, R \rangle$ by $LA.B$, $LA.pc$, $LA.V$ and $LA.adv$, respectively.

If we consider two local aspects asp_1, asp_2 and their local pointcut matches M_1 and M_2 respectively. Weaving asp_1 on its match M_1, using $\text{weave}_\ell(\mathcal{C}_0, asp_1.B, asp_1.V, asp_1.pc, M_1, asp_1.adv)$, results in $\langle \mathcal{C}_1, \mathrm{m}_1 \rangle$. The weaving of asp_2 needs to apply on $\mathcal{C}_1$ and not on $\mathcal{C}_0$ on which $asp_1.pc$ was matched. Therefore its original match M_2 needs to apply to transitions in $\mathcal{C}_1$ as the local weave could change the transitions. To do so we use the transformation function m_1

(see Definition 24) to get the transitions in $\mathcal{C}_1$ from $\mathcal{C}_0$. Therefore the new set of matches in $\mathcal{C}_1$ given M_2 is determined using $\mathrm{newM}(M_2, \mathrm{m}_1)$.

$$\mathrm{newM}(M, \mathrm{m}) = \{\mathrm{m}(t) \mid t \in M\}$$

Since m applies only to one weave, we generalize it to the k^{th} weave by successive application over the $k-1$ weaves using $\langle \mathrm{m}_1 \ldots, \mathrm{m}_{k-1} \rangle$.

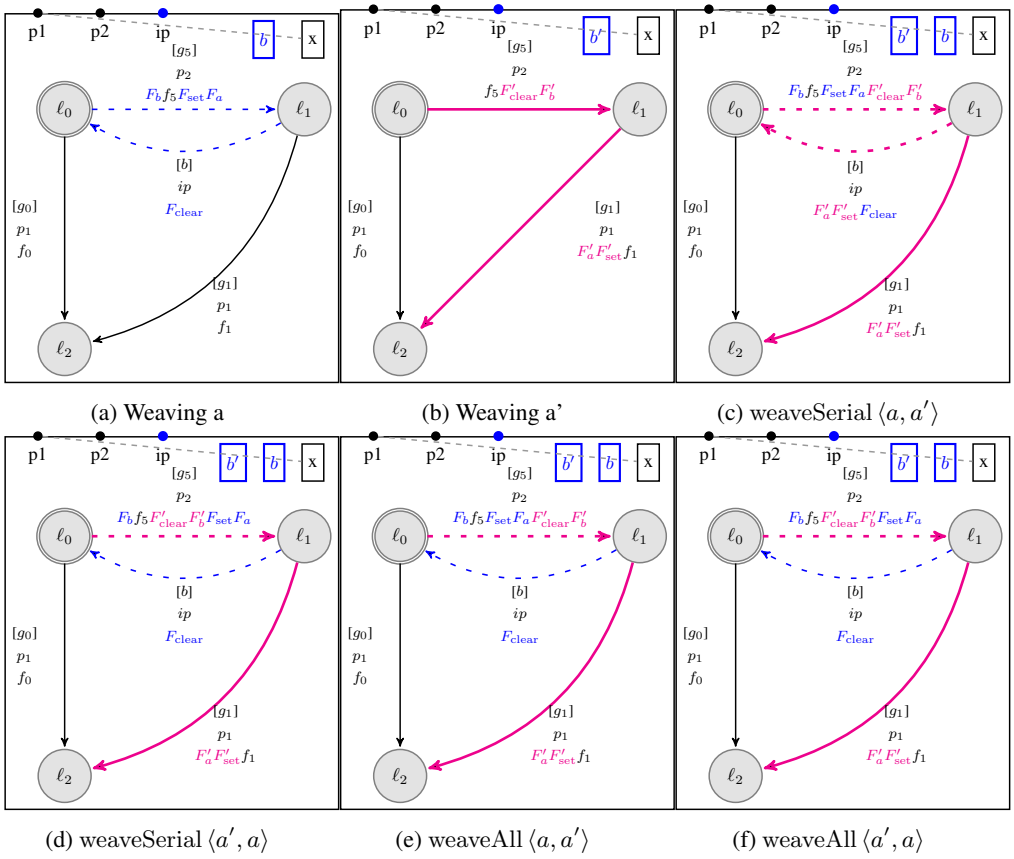
$$\mathrm{follow}(M, \langle \mathrm{m}_1, \ldots, \mathrm{m}_{k-1} \rangle) = \begin{cases} \mathrm{newM}(\mathrm{newM}(\mathrm{newM}(M, \mathrm{m}_1), \ldots), \mathrm{m}_{k-1}) & k-1 > 1 \\ M & \text{otherwise} \end{cases}$$

Definition 27 (weaveAll**).** *The* weaveAll *for a sequence of n local aspects on a composite component $\mathcal{C}_0$ is defined recursively as $\langle \mathcal{C}_n, \mathrm{m}_n \rangle = \mathrm{weaveAll}(\mathcal{C}_0, \langle asp_1, \ldots, asp_n \rangle)$ such that $\forall i \in \{1, \ldots, n\}$:*

$$\langle \mathcal{C}_i, \mathrm{m}_i \rangle = \mathrm{weave}_\ell(\mathcal{C}_{i-1}, follow(M_i, \langle \mathrm{m}_1, \ldots, \mathrm{m}_{i-1} \rangle), asp_i.V, asp_i.adv)$$
$$M_i = \mathrm{select}_\ell(\mathcal{C}_0, asp_i.pc)$$

Example 18. *Figure 8 shows the different results obtained by weaving two local aspects a and a' with the two proposed procedures. The pointcuts are $a.pc = (\mathrm{atLocation}(\ell_0)$ and $\mathrm{portExecute}(p_2))$ and $a'.pc = \mathrm{atLocation}(\ell_1)$. The corresponding advices are $a.adv = \langle F_b, F_a, \{\langle \ell_0, \texttt{true} \rangle\} \rangle$ and $a'.adv = \langle F'_b, F'_a, \{\} \rangle$.*

- *Figure 8a and Fig. 8b show the weaving of each aspect individually. Let M (resp. M') be the match of $a.pc$ (resp. $a'.pc$) and t_5 the transition guarded by g_5, t_5 is both in M and $\mathrm{prev}(M')$. In this case, the advices from a and a' overlap when woven on t_5.*
- *Figure 8c and Fig. 8d illustrate the* $\mathrm{weaveSerial}_\ell$ *operation.*
  1. *Figure 8c presents the serial weave of a followed by a'. The weave of a results in $\langle \mathcal{C}_1, \mathrm{m}_1 \rangle$. $\mathcal{C}_1$ is shown in Fig. 8a. Upon weaving a' its pointcut will match the reset location as it is outbound from ℓ_1 and will therefore prepend F'_a to it. The pointcut will also match $\mathrm{m}_1(t_5)$ as it is inbound, therefore it appends $\langle F'_b, F'_{\mathrm{clear}} \rangle$ to its existing function ($\langle F_b, f_5, F_{\mathrm{set}}, F_a \rangle$) on which a was already woven.*
  2. *Figure 8d presents the serial weave on a' followed by a. The weave of a' results in $\langle \mathcal{C}'_1, g'_1 \rangle$. The component is shown in Fig. 8b. Upon weaving a the pointcut will match $\mathrm{m}'_1(t_5)$. Its function $\langle f_5, F'_b, F'_{\mathrm{clear}} \rangle$ is then appended with $\langle F_a, F_{\mathrm{set}} \rangle$. Additionally, the reset location is then added guarded by $b \wedge \texttt{true}$.*
- *Figure 8f illustrates the* weaveAll *operation and how it differs from* weaveSerial*. The pointcut of a' will always match against the original component, matching always the transition guarded by t_5. The* weaveAll *procedure projects the match. The new match result will be t_5 if a' is woven first or $\mathrm{m}_1(t_5)$ if a is woven first. The joinpoint will therefore not contain the reset location in both cases. The order of the aspects still defines the order of the advices woven on overlapping transitions. Fig. 8e displays the different advice order.*



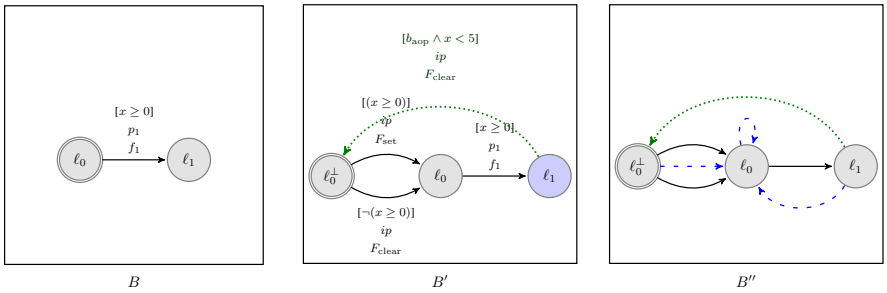
(a) Weaving a (b) Weaving a' (c) weaveSerial $\langle a, a' \rangle$

(d) weaveSerial $\langle a', a \rangle$ (e) weaveAll $\langle a, a' \rangle$ (f) weaveAll $\langle a', a \rangle$

Figure 8: Weave Procedures

Reset location.. In both cases, whenever multiple reset locations are woven on the same location, they are all woven with an extra port appended to the component (the ip port). Since at every weave a singleton interaction has the highest priority, the last local aspect in the sequence has the highest priority on their port. Thus, if multiple reset locations are found on one location (and all their guards evaluate to `true`), the reset location that was woven last will always execute. In the case of weaveSerial, it is possible to match the transitions that originate in the intermediary location, and thus can include different reset locations than expected. Since aspects are assumed to be woven on a new system at every iteration, it is possible to have reset locations on intermediary states created by portEnabled. This could cause undesirable side effects, such as weaving on different locations or transitions (as shown in the following example). By defining more elaborate weaving strategies, one can better manage the expectations of the user and define properties that are preserved or affected by the interference.

Example 19 (Reset location interference). *Figure 9 shows interference when weaving two local aspects a and a' on a base component B (shown in Fig. 9a). The pointcuts expressions for a and a' are respectively :* $\text{portEnabled}(p_1)$ *and* $\text{readVarGuard}(x)$. *Both a and a' use the same reset location set:* $\{\langle \ell_0, x < 5 \rangle\}$. *In this case, we match the transition and create the addi-*



(a) Base (b) Weaving a (c) Weaving a'

Figure 9: Reset Location Interference

tional location $\ell_0^\perp$ as shown in Fig. 9b. However, when matching $\mathrm{readVarGuard}(x)$ *on B', all transitions (including created ones) match since they contain x in their guards. In this case the destination locations are $\ell_0^\perp, \ell_0$ and ℓ_1. Thus, three transitions to ℓ_0 are then created (one from each destination location) when weaving a' as shown in Fig. 9c. In this case we note that the transitions are not created on $\ell_0^\perp$ and execute before the reset location of a. A side effect of the new reset location transitions makes it possible to completely skip evaluating the* $\mathrm{portEnabled}$ *pointcut while moving from $\ell_0^\perp$ to ℓ_0.*

Remark 4 (Strategies and properties). *While we explored specific procedures to weave aspects, and possible issues that are generated from weaving multiple aspects, we stress that for the current work we consider each weave to be on a fresh component. Thus, in our view, the designer, when weaving multiple aspects, is actually looking at the changes in the system from each single weave and determining what should be woven next.*

8. `AOP-BIP`: Aspect-Oriented Programming for BIP Systems

We present `AOP-BIP`, a prototype tool that implements our approach. To test our approach we consider first a network protocol then show the applicability of our approach to monitor CBSs. We begin by describing the network protocol using BIP. Then, we identify cross-cutting concerns and describe them using the `AOP-BIP` language. Finally, we instrument the network model to include these concerns using the `AOP-BIP` tool. We illustrate the weaving of each concern by looking at scattering and tangling. Scattering is measured by counting the elements (affected transitions or interactions in the model) which contain the concern, while tangling is measured by counting the number of elements on which multiple concerns overlap. The scattering and tangling of a concern serve as an indicator of the complexity to implement a concern manually without using `AOP-BIP` tool.

8.1. Tool Overview

`AOP-BIP` is a proof-of-concept, aspect-oriented extension to BIP written in Java ($\sim$ 4300 LOC).

Using the `AOP-BIP` *tool.*. The command-line front end of `AOP-BIP` takes as input:

- A `.bip` file that represents a BIP system written in the BIP language [51];
- The name of the weaving procedure to apply when weaving the aspects on the BIP model;
- A list of `.abip` files that describe the aspects.

`AOP-BIP` produces the BIP model and the aspect containers by parsing the `.bip` file and `.abip` files, respectively. It selects a weaving procedure to compose the aspects *per container* as described in Section 7. `AOP-BIP` then weaves the containers onto the BIP model resulting in an output BIP model.

Overview of the `AOP-BIP` *language.*. The BIP language is described in full in [51]. We use only a subset of the language to illustrate the concepts of this paper. The BIP language is typed. Components, ports and data are associated with types. In addition to a description of the BIP system, `.bip` files also contain a module declaration to encapsulate the system and a `header` section. The header contains arbitrary `C` code that is used during code generation.

The `AOP-BIP` language follows the same ideas presented in this paper with a few extensions.[2] At the file level an `AOP-BIP` file contains a `header` followed by multiple containers. The `header` provides additional `C` code to merge with the `.bip` header. This is useful to define extra functions or include extra libraries. Aspects are grouped into containers. A container is defined by declaring the `Aspect` keyword followed by its identifier[3]. If the container defines local aspects, then it must specify the atomic component it targets right after its identifier. The inter-type variables are not included in the individual aspects, but at the container level. We select an atomic component or an interaction by its *identifier* in the system. Therefore, local aspects apply to a specific instance of the atomic type and global aspects apply to a specific instance of the connector type. The global pointcut syntax includes a port specification: `portspec`. The port specification is used to alias a port identifier. For example, using the specification `p : c1.stop` allows the port `stop` in component `c1` to be referenced as `p` in the rest of the pointcut or the advice, if `c1.stop` has a variable `x`, it can be referenced as `p.x`. This is provided merely as syntactic sugar to simplify referring to the port variables in the read and modified variables in the pointcut and the advice's update functions.

8.2. The `Network` *Example*

A network protocol is used to illustrate the handling of crosscutting concerns in BIP and is shown in Fig. 10. The network protocol is an augmented version of the one presented in [9]. The `Network` composite component consists of a `Server`, a `Client` and a `Channel`. The double circles denote the start locations for each component. The `Server` waits for the *clear-to-send* signal on its `cts` port. This indicates that a channel is available. It then generates a packet and sends it to the channel. The channel forwards the packet to the client which acknowledges it. The channel will then send the acknowledgement back to the server.

[2]The full grammar can be found in Appendix B.

[3]We chose to use `Aspect` instead of `Container` as an implementation decision to have a similar notion than that of AspectJ, since AspectJ defines multiple pairs of pointcut and advice to be an aspect.

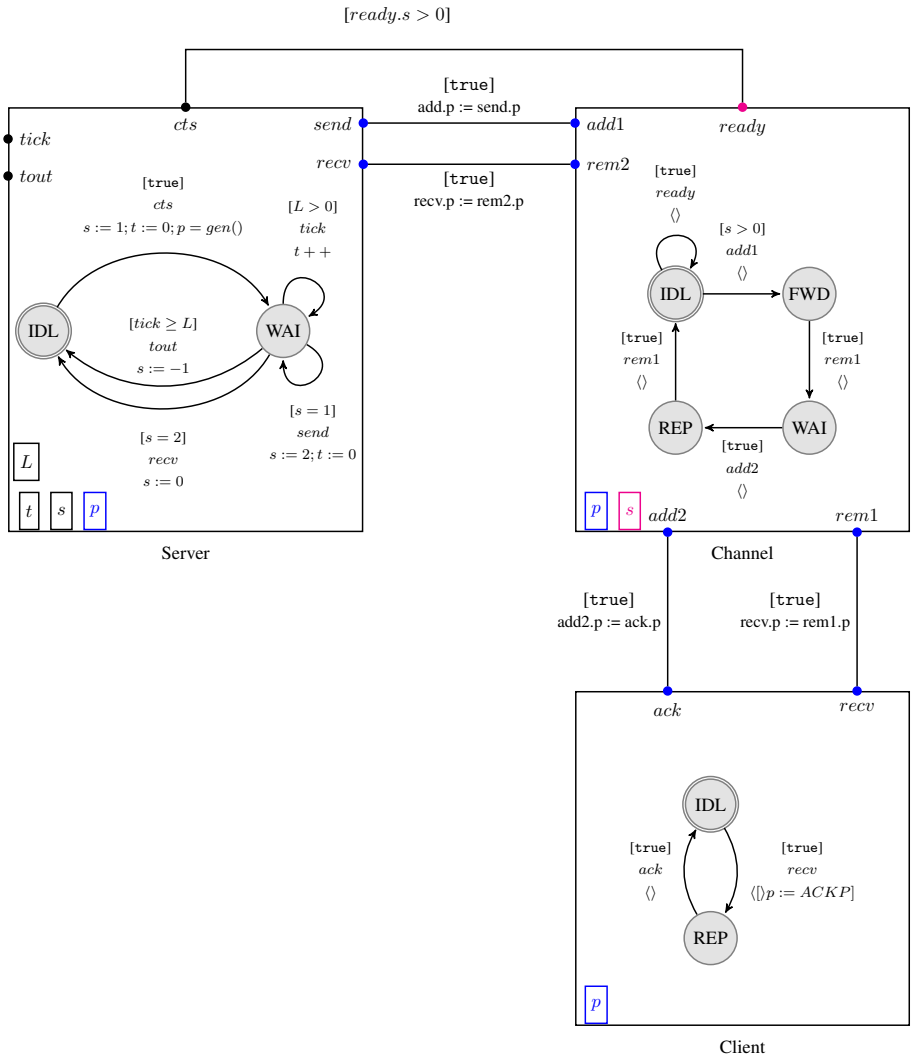


Figure 10: The `Network` Component

Crosscutting concerns.. The network protocol is augmented by refining its specification. The correctness of our transformation ensures that the crosscutting concerns are rigorously handled. First, logging is introduced by capturing port executions locally in all components. Second, security is added in the form of authentication. A signature (hash) is added to the packet and checked. To accomplish the above we introduce two local aspects. The various aspects along with the output are displayed in Fig. 11. The first intercepts the `Server`'s `cts` port execution and adds the signature once the server is ready to send, by modifying the packet stored in the local variable p. The second intercepts the `Channel`'s `add1` port execution; this port executes when a packet from the `Server` is sent. The advice verifies the signature (using `check(p)`) and stores the result (logical 0 or 1) in an inter-type variable `clear`. The advice also adds

```
Aspect AddHash (server)
{
  portExecute(cts)
  do {} {p = wrap(p); }
}
Aspect VerifyHash (channel) {
  data int clear = 0
  portExecute(add1)
  do {} {
    clear = check(p);
    p = unwrap(p);
  }
  {(IDL, clear == 0)}
}
Aspect Carol { // Man-in-the-middle
  ports(a:server.send b:channel.add1)
  readPortVars(a.r)
  do {} {b.r = pfake(a.r);}
}
```

```
[Server.cts  ] Clear to Send
[Server.send ] -> 886|6 (Time: 0)
[Channel.add1] <- 386|6
[Channel.rem1] -> 386
[Client.recv ] <- ACK
[Client.ack  ] -> ACK
[Channel.add2] <- ACK
[Channel.rem2] -> ACK
[Server.recv ] <- ACK (Time: 3)

[Server.cts  ] Clear to Send
[Server.send ] -> 763|3 (Time: 0)
[Channel.add1] <- 736|3
[Server.tout ] Timeout
```

Figure 11: Authentication aspects and resulting execution. The aspects extends the behavior by verifying a hash of the messages and forwarding or blocking the message.

a reset location to `IDL` if the verification failed (`clear` $==$ `0`), preventing the `Channel` from forwarding the packet to the `Client`. The `Carol` aspect is added to modify the packet in transit to display a failed authentication. The global pointcut expression matches: (1) ports `server.send` and `channel.add1`, and (2) variable `server.send.r`. The advice in the `Carol` aspect changes the value of `channel.add1.r` after the execution of any interaction that matches the pointcut. Normally, the system executes the packet transfer by reading `a.r` and modifying `b.r`. The advice function instead will override `b.r` by generating a fake packet from `a.r` using `pfake(a.r)`. The output displays a successful and an unsuccessful attempt. The packet is represented by a number and the signature is the last digit in the number. We notice that the first aspect added `|6` to the packet `886` and the second aspect removed it when the channel forwarded it (`Channel1.rem1`). `Carol` replaces the first packet `886` with `386`, which both have the same signature (`6`). The verification succeeds in this case unlike in the second try, when `763` is replaced by `736` since the signature of `736` is `6` but not `3`. Third, congestion avoidance is added by computing the round-trip time of the message and then waiting before sending further messages. Fourth, basic fault tolerance is introduced in the form of a failsafe mechanism. The system deadlocks and then terminates safely, after the server fails to receive a certain number of acknowledgments.

Coverage of concerns.. The coverage of the concerns is shown in Table 2. Column Transitions reports the number transitions that have been modified including the number of added transitions for reset locations. Column Interactions reports the number of modified interactions. Column OT (resp. OI) reports the number of transitions (resp. interactions) that overlap with other concerns. Interfering concerns are reported in column OC. Concerns are indicated by label (1-4). We illustrate concerns that target multiple areas in the system. Without using `AOP-BIP`, implementing these concerns would require one to edit a significant part of the system. For instance, in the case of logging, the code must be inserted in 10 transitions, of which half overlap with other concerns.

8.3. Applicability to Runtime Verification of CBSs

Overview. Runtime verification (RV) is a lightweight verification technique used to verify whether a run of a specific system verifies a specific property [23, 4, 3]. It consists in extracting a sequence of events, which is then fed to a monitor that verifies it against a specification. RV

Table 2: Crosscutting concerns in `Network`

#	Concern	Transitions	Interactions	OT	OI	OC
1	Logging	10	0	5	0	2,3
2	Authentication	2	1	2	1	1,3.4
3	Congestion	5	0	4	0	1,2
4	Fault Tolerance	0	3	0	1	2
	`Network`	12	5			

frameworks for CBSs, and particularly for BIP systems (RV-BIP [25] and RVMT-BIP [41]) have been already developed. They define specific transformations to instrument components and insert monitors as components in the new system (RV-BIP for sequential systems and RVMT-BIP for multi-threaded systems). However, since runtime verification is a crosscutting concern, it is possible to instrument a system with aspects (both global and local) to generate necessary events for monitoring. At the global level, it is possible to monitor interactions by intercepting their ports and variable accesses. Thus, by describing global pointcuts, we can generate events that are global, and synthesize global aspects that implement monitors. Since we allow for inter-type declarations at the global level, a monitor state can be stored in the inter-type component. The component can then be used to describe a specification for a monitor. At the local level, it is possible to monitor the component state by using local pointcuts. Thus we can generate events that are local to the component. Using local aspects, we can then describe local monitors that are embedded in the component to check for local events. Certain properties however require information from multiple local monitors, thus it is impossible to handle the synchronization with our current approach. Directly writing monitors as aspects is not handled for these types of properties. However, it is possible for each local monitor to print out an event, and a separate monitoring mechanism to verify the entirety offline. While we do not tackle the automatic synthesis of monitors from a specification, we show next how `AOP-BIP` can be used to write manual monitors for specific properties.

Dala robot. A robotic application is used as an example in [25]. The Dala robot [28] is a large and realistic interactive system which consists of a set of modules. Each module is a set of services that corresponds to different tasks and a set of *posters* that are used to exchange data with other modules. A simplified simulation of the modules consists of various services classified as *readers* and *writers* accessing data of a *poster* component and a global clock component `clock`. Reader services read the data in the poster, and writer services modify it. A writer simulates a large data transfer consisting of two phases. In the first phase, the `writer` writes a task id to the poster using an interaction that exchanges the task id between the two ports: `writer.writev`, and `poster.writev`. We assume that the task ids represent an abstract workload assigned to the various services and components of the robot. Upon completion, the interaction consisting of the ports `writer.finishWrite` and `poster.finishWrite` is executed. We use the example to monitor the following properties: mutual exclusion between *writers*, data freshness and ordering of tasks. The monitors are presented in Listing 3.

Monitoring mutual exclusion. The first property focuses on *writers*. The writing process consists of two steps: `write` and `finishWrite`. The `writev` port is triggered when a writer starts writing to the poster service, and upon completion `finishWrite` is executed. The property checks that no two writers are writing at the same time. To monitor the property, we create a local aspect on the `poster` component, and use an extra inter-type variable c initialized to zero. Upon execution of the port `writev`, we check if c is non-zero and increment it by one, and upon execution of the port `finishWrite` we decrement c by one. Thus, if two executions of the port `writev` happen, the monitor can verify the property ($c > 0$).

Monitoring freshness. The second property focuses on *readers*, when a *reader* reads data posted by a *writer*, the freshness property checks for constraints on timestamps. The goal is to make sure that the data being read is up-to-date, i.e. the data has been read at most after a fixed amount of ticks. To do so, the property compares the timestamp stored in the *poster* against the current timestamp stored in the *clock* component, whenever a *reader* reads data. Thus, our monitor considers the interaction consisting of the ports `reader.read`, `poster.read` and `clock.getTime`. The monitor intercepts the port variables to get the timestamp, then computes the difference and verifies whether it is below a threshold of 2 ticks.

Monitoring ordering. The third property concerns the interaction between the *writers* and the *poster*; when a task id is set, the ordering property expresses constraints on the order of tasks. In this example, we check if a task with a larger id executes before one with a smaller id. To do so, we use an inter-type variable `lastTask` and initialize it to zero, indicating that the next expected task is 1. We intercept all the interactions that involve the port `poster.writev` and check for those that write to the port variable (which contains the task id). In this case, the global advice is executed after the call, so as to catch the value written to the port's variable. Upon writing to the variable we check if the new task is in the correct order by comparing it to `lastTask` $+ 1$ and then store the new task id as the `lastTask`.

9. Related Work

Modularizing AOP.. Multiple approaches have sought to (i) improve the applicability of AOP, and (ii) enhance its modularity. These approaches include, but are not limited to Ptolemy [46], XPIs [49], AspectJML [47], and using substitution [40]. They mainly focus on defining contracts and improving matching and advice applications in a modular way, and provide a better decoupling mechanism for the aspects of the systems. These approaches provide us with a perspective on how to better integrate, in a component-based manner, aspects in CBSs. However, they were mainly conceived to improve on the initial AOP methods that target non-CBS systems (i.e., AspectJ [52]). They perceive the system as arbitrary function calls or message passing between objects that is not constrained. As such, they do not inherently target component-based semantics, such as synchronization or data transfers between various modules.

AOP for CBSs.. Pessemier et al. [44] present a framework to deal with crosscutting concerns in CBSs. It is a symmetric approach, i.e. it uses the same language of the system to describe AOP concepts. It presents aspects as components containing the advice and additional interfaces, and are therefore integrated homogeneously within the system. Interaction with the advice is done through these added interfaces. The implementation of a concern is found in what is called an *aspect component*. Aspect components are regular components augmented with extra interfaces.

Listing 3: Monitors

```
{# //Monitoring Functions
  void checkConcurrent(int c) {
    if(c > 0) printf("[Mutex] Violation\n");
  }
  void checkFresh(int clock, int poster) {
    if(clock - poster > 2) {
      printf("[Freshness] Violation: %i - %i = %i\n", clock, poster, condition);
    }
  }
  void checkOrder(int task, int lastTask) {
    if (task != lastTask + 1) {
       printf("[Order] Violation: task %d is preceded by task %d\n",
              task, lastTask);
    }
  }
#}

Aspect MonMutex (poster) {
  data int c = 0;

  portExecute(writev)
  do {checkConcurrent(c); c++;} {}

  portExecute(finishWrite)
  do {}{c--;}
}
Aspect MonFresh {
  ports(r:reader.read p:poster.read c:clck.getTime)
  do  { checkFresh(c.x, p.x); } {}
}
Aspect MonOrder {
  data int lastTask = 0

  ports(p:poster.writev) writePortVars(p.x)
  do {} { checkOrder(p.x, lastTask); lastTask = p.x; }
}
```

They contain the advices necessary to implement a crosscutting concern. Interaction with the advice happens through additional interfaces called *advice interfaces*. Moreover, aspects components expose regular interfaces. Thus, they can be seen as regular components. Joinpoints are a combination of interfaces of different components. Thus, components are seen as black boxes. The interception model is based on composition filters [1] extended from objects to components. The model is mapped onto Fractal, a modular and extensible component model [11]. This approach has several advantages. First it explicitly models dependencies between aspects and components, and allows for their composition at an architectural level. Second, it allows the aspects to be manipulated and reconfigured at runtime. Third, it clearly defines the relationships (1) between aspects and other aspects, and (2) between aspects and the components they modify. This approach, however, does not consider the semantics of interactions. It targets arbitrary interface signatures, so the implementation itself must explicitly address the synchronization amongst the different components and data transfer. Notions of *before*, *after* differ from just executing a function. In the simplest case, a BIP interaction requires ports to be all enabled, therefore *before*

and *after* execute upon synchronization of all involved components.

Similarly, other works such as [19] and [37] integrate AOP into CBSs as well. These approaches are, however, asymmetric (i.e., they use an external language to represent AOP concepts) and subsumed by [44]. Duclos's approach [19] defines two languages to integrate aspects. Lieberherr's approach [37] defines aspects as part of the modules they apply to, and compares the expressiveness of the approach with both AspectJ and HyperJ [43].

SAFRAN [16] differs from the above approaches by using AOP in the Fractal component model to define adaptation policies.

Formalization of aspects.. The aforementioned approaches for CBSs do not rely on formal models. Work to formalize aspects in programs has been undertaken by [31]. The approach specifies different categories of aspects and how they affect various classes of properties (safety, liveness). Aspects are then assigned to a category by syntactic analysis. The work has been extended by Djoko et al. [18] by expanding the categories and defining languages of aspects. The languages of aspects ensure by construction that aspects written with them fit a specific category. Additional tools for verification and analysis of aspects and their interference have been developed and are presented in [32]. The approach focuses on object-oriented systems, and not component-based systems, but it also provides theoretical results on property preservation which our approach does not study. By formalizing aspects in the context of CBS semantics, our approach paves the way to extend these works to CBSs.

Larissa [2] is a language for handling crosscutting concerns in reactive systems modeled as the composition of Mealy automata. The matching is done by assigning monitor programs that look for a specific execution trace. Joinpoints are then associated with the input history. Advices consist of two types: `toInit` and `recovery`. The `toInit` advice places the program back in its original state. The `recovery` advice consists of restoring the program to the last recovery state it was in. Since it is impossible to play the input backwards for recovery, a set of global recovery points is determined. A recovery state is determined by a monitor: the recovery program. The recovery states are associated with specific execution traces and are matched similarity to joinpoints. Compared to our approach, Larissa supports joinpoints based on the input history. It can also be seen as symmetric since aspects are introduced in the synchronous language used. However, the underlying model is conceived for reactive systems, and not CBSs, it does not have a clear distinction between communication and components, and thus does not distinguish between aspects related to components and communications. The communication model is based on simple input/output matching. Moreover, advices are not expressive and only consider reset/restore the state of the system. Formalizing aspects in the BRIC component model has been undertaken by [17]. BRIC formalizes the behavior of components and their interactions using the Communicating Sequential Processes (CSP) language. Unlike our approach, BRIC is symmetric: aspects, pointcuts, and advices are described in CSP, and woven using CSP operators. Additionally, BRIC targets interactions and not the components themselves. It regards components as black boxes. Similarly to BIP, CSP benefits from compositional verification of the properties and has a well-defined semantics. CSP uses denotational semantics as opposed to BIP which uses operational semantics. Verification on the resulting woven system is possible in both approaches. However, BIP has a strong expressive synchronization primitive [8] which is more expressive than CSP [29]. This allows more concerns to be formalized.

10. Conclusions

10.1. Summary

This paper deals with crosscutting concerns in CBS using the AOP paradigm. It targets the two stages in the construction of CBSs by defining local and global aspects to refine components and their compositions, respectively. Local joinpoints capture concerns found in local components. Local pointcuts select joinpoints based on location, guards, variables in update functions and ports. They are associated with advices to add extra functionalities (e.g., computation or location change). Global joinpoints capture the interactions between components through their interfaces without having information about the internal representation of components. Global pointcuts select global joinpoints based on interactions, by considering ports and their respective data transfer operations. They are associated with global advices, to add extra functionality to the interaction model (e.g., data transfer, storing global information, etc.). Furthermore, to remedy interference and to increase expressiveness, we present a way to compose multiple aspects. Global and local joinpoints are mapped to BIP semantics and pointcut matching and advice weaving are implemented using model-to-model transformation on BIP models. We implement the proposed method in the `AOP-BIP` tool-chain. `AOP-BIP` consists of a language to describe both local and global aspects and provides an implementation of matching, weaving and composition. We study the automatic integration of various crosscutting concerns (logging, security, performance, fault handling) on a given input BIP system. Furthermore, we focus on a particular crosscutting concern, namely runtime verification and use `AOP-BIP` to monitor three properties on a robotic application designed as a CBS.

10.2. Future Work

Future work comprises four directions. The first consists in capturing more joinpoints and extending the possible behavior of advices. This would facilitate, in particular, the support for runtime verification [23, 4] and runtime enforcement [21, 22], possibly in a timed setting [14, 6, 45, 26] where the elapsing of physical time influences the behavior of monitors. Possible new joinpoints include variables in interaction guard, specific values of variables. Advices can be extended to modify guards on matching transitions and interactions. The second consists in applying CBS methods to define advices and aspect composition. This would help integrating AOP in BIP symmetrically, where aspects are implemented as components and interactions within the existing system. Moreover, this would allow to enable or disable aspects in the system, and specify more complex advices (i.e., advices as components instead of update functions and extra transitions). The third consists in elaborating new ways to compose aspects by finding new criteria to order them. Aspects can be re-ordered in a container based on their pointcut expressions, by grouping those that affect the same transitions or interactions, and whether or not they modify the existing variables (read/write aspects). Additionally, the language could be extended to allow the explicit definition of precedence rules. The fourth consists in implementing model-to-model transformations using Domain Specific Languages inspired by ATL [30] targeting the BIP model and comparing their expressiveness with our approach.

Acknowledgment

The authors acknowledge the support of the ICT COST (European Cooperation in Science and Technology) Action IC1402 Runtime Verification beyond Monitoring (ARVI) and the University Research Board (URB) at American University of Beirut.

References

[1] Aksit, M., Bergmans, L., Vural, S., 1992. An object-oriented language-database integration model: The composition-filters approach. In: Madsen, O. L. (Ed.), ECOOP'92, European Conference on Object-Oriented Programming.Proceedings. Vol. 615 of Lecture Notes in Computer Science. Springer, Berlin, Germany, pp. 372–395.

[2] Altisen, K., Maraninchi, F., Stauch, D., 2006. Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework. Sci. Comput. Program. 63 (3), 297–320.

[3] Bartocci, E., Falcone, Y. (Eds.), 2018. Lectures on Runtime Verification - Introductory and Advanced Topics. Vol. 10457 of Lecture Notes in Computer Science. Springer.

[4] Bartocci, E., Falcone, Y., Francalanza, A., Reger, G., 2018. Introduction to runtime verification. In: [3], pp. 1–33.

[5] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J., 2011. Rigorous component-based system design using the BIP framework. IEEE Software 28 (3), 41–48.

[6] Bauer, A., Leucker, M., Schallhart, C., 2011. Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. 20 (4), 14:1–14:64.

[7] Bensalem, S., Griesmayer, A., Legay, A., Nguyen, T., Sifakis, J., Yan, R., 2011. D-finder 2: Towards efficient correctness of incremental design. In: Bobaru, M. G., Havelund, K., Holzmann, G. J., Joshi, R. (Eds.), NASA Formal Methods - Third International Symposium, NFM 2011. Proceedings. Vol. 6617 of Lecture Notes in Computer Science. Springer, Berlin, Germany, pp. 453–458.

[8] Bliudze, S., Sifakis, J., 2008. A notion of glue expressiveness for component-based systems. In: van Breugel, F., Chechik, M. (Eds.), Concurrency Theory, 19th International Conference, CONCUR 2008. Proceedings. Vol. 5201 of Lecture Notes in Computer Science. Springer, Berlin, Germany, pp. 508–522.

[9] Bonakdarpour, B., Bozga, M., Gößler, G., 2012. A theory of fault recovery for component-based models. In: Richa, A. W., Scheideler, C. (Eds.), Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012. Proceedings. Vol. 7596 of Lecture Notes in Computer Science. Springer, Berlin, Germany, pp. 314–328.

[10] Bozga, M., Jaber, M., Sifakis, J., 2010. Source-to-source architecture transformation for performance optimization in BIP. IEEE Trans. Industrial Informatics 6 (4), 708–718.

[11] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J., 2004. An open component model and its support in Java. In: Crnkovic, I., Stafford, J. A., Schmidt, H. W., Wallnau, K. C. (Eds.), Component-Based Software Engineering, 7th International Symposium, CBSE 2004. Proceedings. Vol. 3054 of Lecture Notes in Computer Science. Springer, Berlin, Germany, pp. 7–22.

[12] Chen, F., Rosu, G., 2005. Java-MOP: A monitoring oriented programming environment for Java. In: Halbwachs, N., Zuck, L. D. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005. Proceedings. Vol. 3440 of Lecture Notes in Computer Science. Springer, Berlin, Germany, pp. 546–550.

[13] Chkouri, M. Y., Robert, A., Bozga, M., Sifakis, J., 2008. Translating AADL into BIP - application to the verification of real-time systems. In: Chaudron, M. R. V. (Ed.), Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers. Vol. 5421 of Lecture Notes in Computer Science. Springer, pp. 5–19.

[14] Colombo, C., Pace, G. J., Schneider, G., 2009. Safe runtime verification of real-time properties. In: Ouaknine, J., Vaandrager, F. W. (Eds.), Formal Modeling and Analysis of Timed Systems, 7th International Conference, FORMATS 2009, Budapest, Hungary, September 14-16, 2009. Proceedings. Vol. 5813 of Lecture Notes in Computer Science. Springer, pp. 103–117.

[15] Czarnecki, K., Ulrich, E., Steyaert, P., 1997. Beyond objects: Generative programming. In: ECOOP'97 Workshop on Aspect-Oriented Programming. Springer, Berlin, Germany, pp. 5–14.

[16] David, P., Ledoux, T., 2006. An aspect-oriented approach for developing self-adaptive fractal components. In: Löwe, W., Südholt, M. (Eds.), Software Composition, 5th International Symposium, SC 2006, Revised Papers. Vol. 4089 of Lecture Notes in Computer Science. Springer, Berlin, Germany, pp. 82–97.

[17] Dihego, J., Sampaio, A., 2015. Aspect-oriented development of trustworthy component-based systems. In: Leucker, M., Rueda, C., Valencia, F. D. (Eds.), Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium. Proceedings. Vol. 9399 of Lecture Notes in Computer Science. Springer, Berlin, Germany, pp. 425–444.

[18] Djoko, S. D., Douence, R., Fradet, P., 2012. Aspects preserving properties. Sci. Comput. Program. 77 (3), 393–422.

[19] Duclos, F., Estublier, J., Morat, P., 2002. Describing and using non functional aspects in component based applications. In: Proceedings of the 1st International Conference on Aspect-oriented Software Development. AOSD '02. ACM, New York, NY, USA, pp. 65–75.

[20] El-Hokayem, A., Falcone, Y., Jaber, M., 2016. Modularizing crosscutting concerns in component-based systems. In: Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016. Proceedings. Springer, Berlin, Germany, pp. 367–385.

[21] Falcone, Y., 2010. You should better enforce than verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G. J., Rosu, G., Sokolsky, O., Tillmann, N. (Eds.), Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Vol. 6418 of Lecture Notes in Computer Science. Springer, pp. 89–105.
[22] Falcone, Y., Fernandez, J., Mounier, L., 2012. What can you verify and enforce at runtime? STTT 14 (3), 349–382.
[23] Falcone, Y., Havelund, K., Reger, G., 2013. A tutorial on runtime verification. In: Engineering Dependable Software Systems. Vol. 34. IOS Press, Amsterdam, The Netherlands, pp. 141–175.
[24] Falcone, Y., Jaber, M., 2017. Fully automated runtime enforcement of component-based systems with formal and sound recovery. STTT 19 (3), 341–365.
[25] Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S., 2015. Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Software and System Modeling 14 (1), 173–199.
[26] Falcone, Y., Jéron, T., Marchand, H., Pinisetty, S., 2016. Runtime enforcement of regular timed properties by suppressing and delaying events. Systems & Control Letters 123, 2–41.
[27] Filali-Amine, M., Lawall, J. L., 2010. Development of a synchronous subset of AADL. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (Eds.), Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings. Vol. 5977 of Lecture Notes in Computer Science. Springer, pp. 245–258.
[28] Fleury, S., Herrb, M., Chatila, R., 1997. G^{en}om: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In: Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS'97. IEEE, USA, pp. 842–849.
[29] Hoare, C. A. R., 1985. Communicating Sequential Processes. Prentice-Hall, Upper Saddle River, NJ, USA.
[30] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., 2008. ATL: A model transformation tool. Sci. Comput. Program. 72 (1-2), 31–39.
[31] Katz, S., 2006. Aspect categories and classes of temporal properties. In: Transactions on aspect-oriented software development. Springer, Berlin, Germany, pp. 106–134.
[32] Katz, S., Faitelson, D., 2012. The common aspect proof environment. STTT 14 (1), 41–52.
[33] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., 2001. An overview of aspectj. In: Knudsen, J. L. (Ed.), ECOOP 2001 - Object-Oriented Programming, 15th European Conference. Proceedings. Vol. 2072 of Lecture Notes in Computer Science. Springer, Berlin, Germany, pp. 327–353.
[34] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J., 1997. Aspect-oriented programming. In: ECOOP. pp. 220–242.
[35] Kiviluoma, K., Koskinen, J., Mikkonen, T., 2006. Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects. In: Pollock, L. L., Pezzè, M. (Eds.), Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, 2006. ACM, New York, NY, USA, pp. 181–190.
[36] Lieberherr, K. J., Holland, I. M., 1989. Formulations and benefits of the law of Demeter. SIGPLAN Notices 24 (3), 67–78.
[37] Lieberherr, K. J., Lorenz, D. H., Ovlinger, J., 2003. Aspectual collaborations: Combining modules and aspects. Comput. J. 46 (5), 542–565.
[38] Magee, J., Kramer, J., 1996. Dynamic structure in software architectures. In: Garlan, D. (Ed.), SIGSOFT '96, Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, California, USA, October 16-18, 1996. ACM, pp. 3–14.
[39] Milner, R., 1998. The pi calculus and its applications (keynote address). In: Jaffar, J. (Ed.), Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming, Manchester, UK, June 15-19, 1998. MIT Press, pp. 3–4.
[40] Molderez, T., Janssens, D., 2015. Modular Reasoning in Aspect-Oriented Languages from a Substitution Perspective. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 3–59.
[41] Nazarpour, H., Falcone, Y., Bensalem, S., Bozga, M., 2017. Concurrency-preserving and sound monitoring of multi-threaded component-based systems: theory, algorithms, implementation, and evaluation. Formal Asp. Comput. 29 (6), 951–986.
[42] Noureddine, M., Jaber, M., Bliudze, S., Zaraket, F. A., 2014. Reduction and abstraction techniques for BIP. In: Lanese, I., Madelaine, E. (Eds.), Formal Aspects of Component Software - 11th International Symposium, FACS 2014, Revised Selected Papers. Vol. 8997 of Lecture Notes in Computer Science. Springer, Berlin, Germany, pp. 288–305.
[43] Ossher, H., Tarr, P., 2000. Hyper/J: Multi-dimensional separation of concerns for Java. In: Proceedings of the 22Nd International Conference on Software Engineering. ICSE '00. ACM, New York, NY, USA, pp. 734–737.
[44] Pessemier, N., Seinturier, L., Duchien, L., Coupaye, T., 2008. A component-based and aspect-oriented model for

software evolution. IJCAT 31 (1/2), 94–105.
[45] Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena-Timo, O., 2014. Runtime enforcement of timed properties revisited. Formal Methods in System Design 45 (3), 381–422.
[46] Rajan, H., Leavens, G. T., 2008. Ptolemy: A language with quantified, typed events. In: Vitek, J. (Ed.), ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings. Vol. 5142 of Lecture Notes in Computer Science. Springer, pp. 155–179.
[47] Rebêlo, H., Leavens, G. T., Bagherzadeh, M., Rajan, H., Lima, R., Zimmerman, D. M., Cornélio, M., Thüm, T., 2014. Aspectjml: Modular specification and runtime checking for crosscutting contracts. In: Proceedings of the 13th International Conference on Modularity. MODULARITY '14. ACM, New York, NY, USA, pp. 157–168.
[48] Sifakis, J., 2005. A framework for component-based construction extended abstract. In: Aichernig, B. K., Beckert, B. (Eds.), Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany. IEEE Computer Society, pp. 293–300.
[49] Sullivan, K., Griswold, W. G., Rajan, H., Song, Y., Cai, Y., Shonle, M., Tewari, N., Sep. 2010. Modular aspect-oriented design with xpis. ACM Trans. Softw. Eng. Methodol. 20 (2), 5:1–5:42.
[50] Szyperski, C., 1998. Component Software: Beyond Object-oriented Programming. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.
[51] Verimag, 2017. BIP Tools.
URL http://www-verimag.imag.fr/BIP-Tools,93.html
[52] Xerox Coporation, 2003. The AspectJ (TM) Programming Guide.
URL https://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

Appendix A. Proofs

The common assumption for the proofs that include the advice F_b and F_a functions, is that F_b and F_a can be uniquely determined and are not empty. To ensure this, one can add code markers at the start and end of each F_b and F_a which have no effect and are not present in the original system.

PROOF (OF PROPOSITION 1). We consider a global event $\langle q, a, q' \rangle \in \mathcal{E}$ and a global pointcut expression $\langle p, v_r, v_w \rangle$. The proof follows from the definition of $\mathrm{select_g}(\mathcal{C}, gpc)$ which selects all interactions matching the criteria that should be matched by $(E_i = \langle q, a, q' \rangle \vDash gpc)$.

$$
\begin{aligned}
(a \vDash \langle p, v_r, v_w \rangle) \text{ iff } & p \subseteq a.ports \\
& \wedge\, v_r \subseteq \mathrm{readvar}(a.func) \\
& \wedge\, v_w \subseteq \mathrm{writevar}(a.func) && \text{(Definition 14)} \\
\text{iff } & a \in \mathrm{select_g}(\gamma, gpc) && \text{(Def. of } \mathrm{select_g}\text{)}
\end{aligned}
$$

PROOF (OF PROPOSITION 2). Given a global advice $\langle F_b, F_a \rangle$. We consider an executed interaction $a \in \mathcal{E}'$, and $\mathrm{rem_g}(a, F_b, F_a) = a'$ the constructed interaction without the advice.

$$
\begin{aligned}
\exists F : a.func = \langle F_b, F, F_a \rangle \text{ iff } & \exists a' \in \gamma : \mathrm{m}(a') = a \wedge a' \in \mathcal{I} && \text{(Definition 18, m())} \\
\text{iff } & a' \in \mathrm{select_g}(\mathcal{C}, gpc) && \text{(Definition 18)} \\
\text{iff } & (\langle q'_s, a', q'_e \rangle \vDash gpc) && \text{(Proposition 1)} \\
\text{iff } & \mathrm{rem_g}(a, F_b, F_a) \vDash gpc
\end{aligned}
$$

An executed interaction's update function $a.func$ starts with F_b and ends with F_a, according to the definition of $\mathrm{m}()$ in Definition 18 iff it is the result of weaving the advice on it from an interaction a' ($\exists a' \in \gamma : \mathrm{m}(a') = a \wedge a' \in \mathcal{I}$). The interaction a' is in $\mathcal{I}$ iff it was selected by

the local pointcut expression ($a' \in \text{select}_{\text{g}}(\mathcal{C}, gpc)$). According to Proposition 1, any interaction $a' \in \text{select}_{\text{g}}(\mathcal{C}, gpc)$ is a joinpoint, specifically $a' = \text{rem}_{\text{g}}(a, F_b, F_a)$.

PROOF (OF PROPOSITION 3). Let $e = \langle\langle l, v\rangle, \tau, \langle l', v'\rangle\rangle$, we assume that lpc does not contain $\text{portEnabled}(p)$. The proof follows by induction on the structure of lpc.

$$e \vDash lpc \text{ iff } \tau \in \text{select}_{\ell}(B_k, lpc)$$

Base cases:

$$\begin{aligned}
e \vDash \text{atLocation}(\ell) &\text{ iff } l = \ell && \text{(Definition 19)}\\
&\text{iff } \tau.src = \ell\\
&\text{iff } \tau \in \{t \in B_k.trans \mid t.src = \ell\}\\
&\text{iff } \tau \in \text{select}_{\ell}(B_k, \text{atLocation}(\ell)) && \text{(Definition 20)}\\
e \vDash \text{readVarGuard}(x) &\text{ iff } (\exists t \in B_k.trans : t.src = l = \tau.src\\
&\wedge x \in \text{readguard}(t)) && \text{(Definition 19)}\\
&\text{iff } \tau \in \text{siblings}(t) && \text{(Def. siblings)}\\
&\text{iff } \tau \in \text{select}_{\ell}(B_k, \text{readVarGuard}(x)) && \text{(Definition 20)}\\
e \vDash \text{readVarFunc}(x) &\text{ iff } (x \in \text{readvar}(\tau.func)) && \text{(Definition 19)}\\
&\text{iff } \tau \in \{t \in B_k.trans \mid x \in \text{readvar}(t.func)\}\\
&\text{iff } \tau \in \text{select}_{\ell}(B_k, \text{readVarFunc}(x)) && \text{(Definition 20)}\\
e \vDash \text{portExecute}(p) &\text{ iff } (\tau.port = p)) && \text{(Definition 19)}\\
&\text{iff } \tau \in \{t \in B_k.trans \mid t.port = p\}\\
&\text{iff } \tau \in \text{select}_{\ell}(B_k, \text{portExecute}(x)) && \text{(Definition 20)}
\end{aligned}$$

$e \vDash \text{write}(x)$ iff $\tau \in \text{select}_{\ell}(B_k, \text{write}(x))$ is shown by replacing in the above: $\text{readVarFunc}(x)$ with $\text{write}(x)$.

Inductive step:

$$\begin{aligned}
e \vDash \phi \text{ and } \phi' &\text{ iff } e \vDash \phi \wedge e \vDash \phi'\\
&\text{iff } \tau \in \text{select}_{\ell}(B_k, \phi) \wedge \tau \in \text{select}_{\ell}(B_k, \phi') && \text{(Hypothesis)}\\
&\text{iff } \tau \in \text{select}_{\ell}(B_k, \phi) \cap \text{select}_{\ell}(B_k, \phi')
\end{aligned}$$

PROOF (OF PROPOSITION 4). We consider F_b and F_a to be the advice before and after update functions, and the local event $e_i \in T'_k$ and $e' = \text{rem}_k(e_i, F_b, F_a)$ to be the constructed event without the advice. The proof is split into two parts: the first proves the correct application of an advice's update functions, the second proves the correct application of reset locations.

.

$$(e'_i \neq \epsilon \wedge e'_i \vDash lpc) \text{ iff } \text{before}(e_i, F_b, d) \wedge \text{after}(e_i, F_a, d)$$

We distinguish two cases based on $\text{early}(lpc)$.

Case 1: $\text{early}(lpc) = \texttt{false}$. This case includes the edit frames $\langle CB, CE\rangle, \langle RUN, CE\rangle$. $\text{before}(e_i, F_b, \texttt{false}) \wedge \text{after}(e_i, F_a, \texttt{false})$ simplifies to $\exists F : e_i.\tau.func = \langle F_b, F, F_a\rangle$.

Case 1.1. We consider that $\mathrm{portEnabled}$ is not included in *lpc*, then the advice is woven on the update function F, $e_i.\tau = \langle \ell, p, g, F_b F F_{\mathrm{set}} F_a, \ell' \rangle$ iff it is the result of a matching transition $\exists t : t = \langle \ell, p, g, F, \ell' \rangle \in M$ according to $\mathrm{wframe}_{\mathrm{cur}}$ in Definition 24 with $e_i.\tau \in \mathrm{m}(t)$. We have $t = \mathrm{rem}_k(e_i, F_b, F_a).\tau$ and $t \in \mathrm{select}_\ell(B_k, \mathit{lpc})$ iff $\mathrm{rem}(e_i, F_b, F_a) \vDash \mathit{lpc}$ from Proposition 3.

Case 1.2. We consider that $\mathrm{portEnabled}$ is included in *lpc*, the set of enabled ports in the expression is $P = \mathrm{sp}(\mathit{lpc})$. Using the definition of $\mathrm{wframe}_{\mathrm{runa}}$ in Definition 24, the advice is woven on the transition $e_i.\tau = \langle \ell, p, b_{\mathrm{aop}} \wedge g, \langle F_b, F, F_a \rangle, \ell' \rangle$ iff $\exists t : e_i.\tau \in \mathrm{m}(t)$ with $t = \langle \ell, p, g, F, \ell' \rangle = \mathrm{rem}_k(e_i, F_b, F_a).\tau$. We decompose $\mathrm{rem}_k(e_i, F_b, F_a) \vDash \mathit{lpc}$ into two conditions $\mathrm{rem}_k(e_i, F_b, F_a) \vDash c_1 \wedge \mathrm{rem}_k(e_i, F_b, F_a) \vDash c_2$, where c_1 contains the conjunction of all the $\mathrm{portEnabled}$ pointcuts and c_2 contains all the rest (as per Definition 19).

1. $e'_i.\tau \in \mathrm{select}_\ell(B_k, \mathit{lpc})$ iff $e'_i \vDash c_2$ by applying Proposition 3 to the syntactic part.
2. $e_i.\tau$ executed iff b_{aop} is `true`. b_{aop} is `true` iff $\mathrm{mkGuard}(P, \ell, M)$ since the instrumentation guarantees that the only transitions that set b_{aop} are guarded by $\mathrm{mkGuard}$. M contains all the transitions outbound from ℓ since $\mathrm{portEnabled}$ uses $\mathrm{siblings}$ (Definition 20). However, by the definition of $\mathrm{mkGuard}$, $\mathrm{mkGuard}$ iff $\forall p_j \in P, \exists \langle \mathrm{rem}_k(e_i, F_b, F_a).l : p_j, g_j, f_j, \ell_j \rangle$ with $g_j(\mathrm{rem}_k(e_i, F_b, F_a).v) = \mathtt{true}$. Therefore, $\mathrm{mkGuard}$ iff $e'_i \vDash c_1$.

Therefore, we have:

$$\mathrm{before}(e_i, F_b, \mathtt{false}) \wedge \mathrm{after}(e_i, F_a, \mathtt{false}) \text{ iff } e'_i \neq \epsilon \wedge e'_i \vDash \mathit{lpc}$$

Case 2: $\mathrm{early}(\mathit{lpc}) = \mathtt{true}$. This case includes the edit frames $\langle PE, CB \rangle, \langle RUN, CB \rangle$. $\mathrm{before}(e_i, F_b, \mathtt{true}) \wedge \mathrm{after}(e_i, F_a, \mathtt{true})$ simplifies to $\exists F, F' : e_{i-1}.\tau.\mathit{func} = \langle F, F_b \rangle \wedge e_i.\tau.\mathit{func} = \langle F_a, F \rangle$. The proof is similar to Case 1, we assume $i > 0$ for simplicity and decompose into two cases: *lpc* does not contain (resp. contains) $\mathrm{portEnabled}$ and use the definition of $\mathrm{wframe}_{\mathrm{prev}}$ and $\mathrm{wframe}_{\mathrm{runb}}$ from Definition 24 respectively. $e_i.\tau$ corresponds to a $t \in M$. Since we consider the event before $e_{i_1}.\tau$ we note that both frames weave on all transitions that lead to M. In the case of $\mathrm{wframe}_{\mathrm{prev}}$, F_b is woven at the end of all transitions in $P(M) \setminus M$ and in the case of the loop from the created locations to the locations from which M is outbound therefore including all possible previous transitions. In the case of $\mathrm{wframe}_{\mathrm{runb}}$, F_b is woven at the created transition guarded by $\mathrm{mkGuard}$ (which is `true`).

.

$$(e'_i \neq \epsilon \wedge e'_i \vDash \mathit{lpc}) \implies (R \neq \emptyset \implies \exists r \in R : \mathrm{reset}(e_i, r)$$

Since it is possible to constructs e'_i and e'_i is a joinpoint, we decompose $e'_i \vDash \mathit{lpc} = e'_i \vDash c_1 \wedge e'_i \vDash c_2$ into a conjunction of two conditions (similarly to Case 1.2). c_1 is a conjunction of all $\mathrm{portEnabled}$ pointcuts and c_2 all the rest.

1. Since $e'_i \vDash c_1$, we have $\forall p_j \in \mathrm{sp}(\mathit{lpc}) : \exists \tau' = \langle e'_i.l, p_j, g_{\tau'}, f_{\tau'}, \ell'_{\tau'} \rangle$ by Definition 19 with $e'_i.\tau \in \mathrm{siblings}(\tau')$ (definition of $\mathrm{siblings}$). Therefore, $e'_i.\tau \in \mathrm{select}_\ell(B_k, c_1)$
2. From Proposition 3, $e'_i \vDash c_2$ iff $e'_i.\tau \in \mathrm{select}_\ell(B_k, c_2)$.

We have from the above

$$e'_i.\tau \in \mathrm{select}_\ell(B_k, c_1) \wedge e'_i.\tau \in \mathrm{select}_\ell(B_k, c_2) \text{ iff } e'.\tau \in (\mathrm{select}_\ell(B_k, c_1) \cap \mathrm{select}_\ell(B_k, c_2)).$$

Therefore $e'_i \vDash lpc$ iff $e'_i.\tau \in \text{select}_\ell(B_k, lpc)$ (i.e., $e'_i.\tau \in M$ and $e_i.\tau \in \text{m}(e'_i.\tau)$). The local weave guarantees that since $e'_i.\tau \in M$ then F_{set} has been called either on $e_i.\tau$ in the case where portEnabled is not present, or $e_{i-1}.\tau$ (F_{set} is in the update function of the created transition precedes $e_i.\tau$ guarded by mkGuard). The local weaving (Definition 24) ensures for all cases of edit frames, that the transitions will lead to a location on which reset location is woven (by using $\text{dest}(M)$ and adding the appropriate created locations). Therefore, it ensures that the next location after $e_i.\tau$ executes, contains all reset location pairs if $R \neq 0$. Using the definition of wReset (Definition 24), for any $r_j = \langle g_j, dest_j \rangle \in R$, we have a transition $reset_j = \langle e_{i+1}.l, ip, b_{\text{aop}} \wedge g_j, f_j, dest_j \rangle$. The port ip corresponds to an interaction with the highest priority, therefore if ip is enabled, i.e., $b_{\text{aop}} \wedge g_j(e_{i+1}.v)$ then it will execute with higher priority than other transitions on the location. Since $b_{\text{aop}} = \texttt{true}$, if $g_i(e_{i+1})$ then the component will execute the reset location transition and move to $dest_j$ (i.e., $e_{i+1}.l' = dest_j$).

Appendix B. `AOP-BIP` Language

The `AOP-BIP` language (overviewed in Section 8.1) follows the ideas presented in this paper. Because of the differences between the global and the local view, the grammar distinguishes global from local pointcut expressions and advices. Furthermore, the identifiers specified in the AOP model generated from an `.abip` file are expected to reference identifiers in the BIP model generated from the `.bip` file. The `validators` modules (both local and global) in the tool are responsible for verifying that the identifiers match.

Listing 4: The AOP-BIP Language Grammar

```
file       : (CODE)? (container)+  ;
container  : 'Aspect' IDENTIFIER '{' intertype (gaspect)+  '}'
           | 'Aspect' IDENTIFIER '(' IDENTIFIER ')'
              '{' intertype (aspect)+ '}'
           ;
CODE       : '{#' .*? '#}';
intertype  : (intertypedef)*
           ;
intertypedef
           : 'data' IDENTIFIER IDENTIFIER
           | 'data' IDENTIFIER IDENTIFIER '=' literal_expression
           ;
aspect     : pointcuts 'do' advice;
pointcuts  : (pointcut)+ ;
pointcut   : pctype '(' IDENTIFIER ')';

pctype     : 'atLocation' |'readVarGuard' | 'readVarFunc'
           | 'write'      | 'portEnabled' | 'portExecute'
           ;
advice     : (before) (after) (resetlocs)? ;
before     : '{' actions '}';
after      : '{' actions '}';
resetlocs  : '{' (rlocpair (',' rlocpair)*)? };
rlocpair   : '(' IDENTIFIER ',' expression  );
gaspect    : gpoint 'do' before after
           ;
gpoint     : 'ports' '(' (portspec)+ ')' (gread)? (gwrite)?
           | 'ports' '(' (portspec)+ ')' (gwrite)? (gread)?
           ;
gwrite     : 'writePortVars' '(' port_var+ ')';
gread      : 'readPortVars'  '(' port_var+ ')';
portspec   : IDENTIFIER ':' port_name ;
port_name  : IDENTIFIER '.' IDENTIFIER;
port_var   : IDENTIFIER '.' IDENTIFIER;
```