AMERICAN UNIVERSITY OF BEIRUT

A GENERAL FRAMEWORK FOR THE INTEGRATION OF CROSSCUTTING CONCERNS IN BIP

by
ANTOINE PIERRE EL HOKAYEM

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon

AMERICAN UNIVERSITY OF BEIRUT

A GENERAL FRAMEWORK FOR THE INTEGRATION OF CROSSCUTTING CONCERNS IN BIP

by

ANTOINE PIERRE EL HOKAYEM

Approved by:

[illegible]

Dr. Mohamad Jaber, Assistant Professor — Advisor
Computer Science

[illegible]

Dr. Paul Attie, Associate Professor — Member of Committee
Computer Science

[illegible]

Dr. Haidar Safa, Associate Professor — Member of Committee
Computer Science

Date of thesis defense: August 24, 2015

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: EL HOKAYEM ANTOINE PIERRE

EL HOKAYEM	ANTOINE	PIERRE
Last	First	Middle

[✓ Master's Thesis] ○ Master's Project ○ Doctoral Dissertation

☒ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, **three years after the date of submitting my thesis, dissertation, or project,** to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

[illegible] 25-08-2015

Signature Date

ACKNOWLEDGEMENTS

Special thanks as to my advisor, Prof. Mohamad Jaber, for his supervision, availability and assistance, and to Prof. Yliès Falcone for his added insight and review.

The thesis has received funding from the University Research Board (URB) at AUB.

AN ABSTRACT OF THE THESIS OF

Antoine Pierre El Hokayem for Master of Science
Major: Computer Science

Title: A General Framework for the Integration of Crosscutting Concerns in BIP

Most computer systems almost certainly harbor undetected errors because of the gap between requirements and implementation. In this thesis, we define a method that combines Aspect Oriented Programming (AOP) and Component-based Systems (CBSs). AOP is a programming paradigm aiming at supporting the separation of concerns during the development of monolithic systems. We use the Behavior Interaction Priority (BIP) framework which is a component-based framework with formal operational semantics. We distinguish two types of aspects: Local and Global. Local aspects are used to model concerns to refine components. Global aspects are used to 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. Our combination of AOP and CBS yields a CBS framework with formal semantics and rigorous transformation primitives. Our method is fully implemented and tested on non-trivial examples.

Contents

ACKNOWLEDGEMENTS v

ABSTRACT vi

LIST OF FIGURES x

LIST OF TABLES xi

1 INTRODUCTION 1
- 1.1 Problem Definition 1
- 1.2 Our Approach 2
- 1.3 Thesis Organization 3

2 BEHAVIOR INTERACTION PRIORITY (BIP) 4
- 2.1 BIP Framework 4
- 2.2 Atomic Components 4
- 2.3 Interactions 6
- 2.4 Priorities 8
- 2.5 BIP System 9
- 2.6 BIP Toolchain 10

3 ASPECT ORIENTED PROGRAMMING 12
- 3.1 Crosscutting Concerns 12
- 3.2 AOP Concepts 14
  - 3.2.1 Joinpoints 14
  - 3.2.2 Pointcuts 15
  - 3.2.3 Advice 16
  - 3.2.4 Aspect 17
  - 3.2.5 Weaving 17
- 3.3 Overview of applying AOP in BIP 18

4 LOCAL ASPECTS . . . 20
4.1 Preliminaries . . . 21
4.2 Local Joinpoints . . . 22
4.3 Local Pointcuts . . . 24
4.3.1 Location . . . 25
4.3.2 Variable Access . . . 26
4.3.2.1 Variable Read Guard . . . 26
4.3.2.2 Variable Read/Write Function . . . 27
4.3.3 Ports . . . 28
4.3.3.1 Port Execution . . . 28
4.3.3.2 Port Enablement . . . 29
4.3.4 Composing Pointcut Expressions . . . 30
4.4 Local Advice . . . 32
4.5 Local Aspect . . . 33
4.6 Weaving Local Aspects . . . 34
4.6.1 Strategy . . . 34
4.6.2 Weaving $\langle CB, CA\rangle$ . . . 35
4.6.3 Weaving $\langle PA, CB\rangle$ . . . 36
4.6.4 Weaving $\langle ADD, CB\rangle$ . . . 37
4.6.5 Weaving Reset Location Pairs . . . 40
4.6.6 Weaving the Local Aspect . . . 40

5 GLOBAL ASPECTS . . . 45
5.1 Global Joinpoints . . . 45
5.2 Global Pointcuts . . . 46
5.3 Global Advice . . . 48
5.4 Global Aspect . . . 49
5.5 Weaving . . . 49

6 ENCAPSULATING ASPECTS . . . 53
6.1 Interference . . . 53
6.2 Containers . . . 54
6.3 Weaving Procedures . . . 55

7 AOP-BIP . . . 60
7.1 Overview . . . 60
7.2 Language Description . . . 61
7.3 Frontend . . . 66
7.4 Backend . . . 68
7.5 Utility . . . 71

8 CASE STUDY 73
8.1 Overview 73
8.2 Logging Concern 74
8.3 Security Concerns 75
8.4 Performance Concerns 76
8.5 Fault Tolerance Concerns 77
8.6 Using Inter-type Structures 77

9 RELATED WORK 78
9.1 Aspect-Oriented and Component-Based Design Integration 78
9.2 Integration of Aspect-Oriented concepts in Automata 81

10 CONCLUSION AND FUTURE WORK 82
10.1 Conclusion 82
10.2 Future Work 83

List of Figures

2.1 Example Atomic Component 6
2.2 Composite Component 8
2.3 BIP Tool-Chain 10

3.1 Multiple Concerns in a Simple System 13
3.2 Local and Global Views 19

4.1 Origin, Destination, and Siblings 21
4.2 Identifying Joinpoints in Atomic Components 22
4.3 Execution Ordering in Atomic Components 24
4.4 Matching $atLocation(\ell_2)$ 25
4.5 Matching $readVarGuard(x)$ 27
4.6 Matching Function Variables 28
4.7 Matching Ports 29
4.8 Pointcuts and Joinpoint Frames 30
4.9 Combining Pointcut Matches 31
4.10 Applying Advice to $atLocation(\ell_2)$ 33
4.11 Weaving $\langle CB, CA \rangle$ 36
4.12 Weaving $\langle PA, CB \rangle$ 38
4.13 Weaving $\langle ADD, CB \rangle$ 41
4.14 Weaving Reset Locations 42

5.1 Example Composite Component 46
5.2 Matching Global Pointcuts 47
5.3 Inter-type Weaving 50
5.4 Completing The Weave 51

6.1 Interference In Atomic Components 54
6.2 Weaving Procedures 58

7.1 The AOP-BIPTool 60
7.2 The Aspect Interface 67
7.3 The Pointcut Interface 69

8.1 The `Network` Component 74

9.1 Aspect Components, Aspect Bindings and Aspect Domains 79

List of Tables

2.1 Possible enabled ports at ℓ_0 6
2.2 Multiple Enabled Interactions 9

4.1 Summary of Pointcut Match Frames 35

Chapter 1
INTRODUCTION

Contents

1.1 Problem Definition 1
1.2 Our Approach 2
1.3 Thesis Organization 3

1.1 Problem Definition

On the one hand, one of the most sensible techniques to tackle complex and large problems is to decompose them into smaller ones. The component-based approach [1] consists in building complex systems by composing components (building blocks). This confers numerous advantages (e.g., productivity, incremental construction, compositionality) that allow to deal with complexity in the construction phase. Component-based systems (CBSs) are desirable because they allow reuse of sub-systems as well as their incremental modification without requiring global changes. Their development requires methods and tools supporting a concept of architecture which characterizes the coordination between components. An architecture structures a system and involves components and relationships between the externally visible properties of those components. The global behavior of a system can, in principle, be inferred from the behavior of its components and its architecture. Component-based design is based on the separation between coordination and computation. Systems are built from units processing sequential code insulated from concurrent execution issues. The isolation of coordination mechanisms allows a global treatment and analysis on coordination constraints between components even if local computations on components are not visible (i.e., components are "black boxes").

On the other hand, Aspect Oriented Programming (AOP) [2, 3] is a programming paradigm aiming at supporting the *separation of concerns* [4] during the development of monolithic systems. A concern is defined in [5] as a "*domain used as a decomposition criterion for a system or another domain with that concern*". Such different domains could include logging, security, persistence, maintenance and synchronization. Concerns are often found in different parts of a system, or in some cases multiple concerns overlap one region. AOP 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. Moreover, AOP uses *weaving rules* to specify how to integrate concerns in order to build the final system.

Developers must address all requirements in order to satisfy the overall system goal. For this, validation and verification (V&V) are paramount during system development. From an abstract point of view, V&V consist of proving that the delivered system satisfies the given requirements. Nonetheless, the system almost certainly harbors undetected errors. This is mainly due to the gap between requirements and implementations.

To tackle the aforementioned issues, we need a framework that combines (1) component-based approach in order to specify the core of the system; and (2) well-defined cross-cutting concerns that refine the system to build the final implementation.

We use the Behavior Interaction Priority (BIP) framework [1, 6, 7] which is a component-based framework with formal operational semantics. Coordination between components is achieved by using multiparty interactions and dynamic priorities for scheduling interactions. BIP consists of three layers: (1) behavior which is handled by atomic components; (2) Interaction that describe the collaboration between the atomic components; (2) Priority chooses which interaction to execute out of many. BIP is used to specify CBSs and is also capable of efficient code generation.

In case of CBSs, crosscutting concerns arise at the level of components [8, 9] (building blocks) and architectures (communications). Integrating crosscutting concerns in CBSs improves the progressivity of building complex systems by refining its core specification. More importantly, it allows users to separately reason about crosscutting concerns.

1.2 Our Approach

Our approach targets to combine the BIP framework with the aspect oriented paradigm. We first seek to 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. Once concerns are formalized, we determine the *rules* that govern the integration of these concerns in the BIP model. Therefore, given an existing BIP system, and a description of concerns, we are to transform the model so as to include the desired concerns. We distinguish two types of aspects: *Local* and *Global.* Local aspects are used to model crosscutting concerns within components. Hence, they are used to refine the behaviors of components. On the other hand, global aspects are used to model crosscutting concerns at the architecture level. Hence, they are used to refine communications (synchronization and data transfer) between components. We additionally define aspect containers which serve as constructs for grouping aspects. We formalize the composition of aspects and their integration into a BIP system. Moreover, we define a high-level language for writing local and global aspects as well as aspect containers. Our framework is fully implemented and tested on non-trivial examples.

Note that, there exists two approaches to integrate AOP into CBS: symmetric and asymmetric [10]. A symmetric approach presents aspects as components, and are therefore integrated homogeneously within the existing components in the system. An asymmetric approach uses a different representation for aspects than the system itself. Our approach is asymmetric, aspects are described using a different representation than BIP models.

1.3 Thesis Organization

The thesis consists of 10 chapters. Chapter 2 introduces the context and the problem, along with an overview of the proposed approach. Chapter 3 presents the BIP framework. Chapter 4 presents the concepts used in AOP. Chapters 5 and 6 define local and global aspects, respectively. Chapter 7 defines the composition of aspects. Chapter 8 described AOP-BIP, the tool implementing our method. Chapter 9 presents some experimental results on a case study. Chapter 10 presents some related work. Finally, Chapter 11 draws conclusions and presents future work.

Chapter 2
BEHAVIOR INTERACTION PRIORITY (BIP)

Contents

2.1 BIP Framework 4

2.2 Atomic Components 4

2.3 Interactions 6

2.4 Priorities 8

2.5 BIP System 9

2.6 BIP Toolchain 10

2.1 BIP Framework

The Behavior Interaction Priority (BIP) framework [1, 6, 7] presents a system as a set of atomic components with prioritized interactions. Atomic components are Labelled Transition Systems (LTS) extended with data. The atomic components define their own interface by exposing a set of ports. Interactions are defined over ports and can be used for synchronization and data transfer between components. The interactions are prioritized based on the priority layer.

2.2 Atomic Components

Definition 1 (Port). *A port $\langle p, x_p \rangle$ in an atomic component B is identified by a port identifier p and a set of attached local variables x. We denote a port by its identifier p and its variables by $p.vars$.*

An atomic component behaves as an LTS extended with data.

Definition 2 (Atomic component). *An atomic component B is defined as a tuple $\langle P, L, T, X \rangle$, where:*

- *X is a set of variables.*
- *L is a set of control locations*
- *P is the set of ports such that $\forall p \in P : p.vars \subseteq X$.*

- *$T = L \times P \times \mathbb{B}[X] \times Exp[X] \times L$ is the set of transitions, where $\mathbb{B}[X]$ is the set of boolean predicates over X and $Exp[X]$ is the set of single variable assignment statements in the form $x := f(X)$ such that $x \in X$.*

A transition $\tau \in T$ is denoted by $\langle \ell, p, g, f, \ell' \rangle$ where:

- ℓ is the source location, ℓ' is the destination location;
- p is a port that is exposed by the component;
- g is the guard, a boolean function over X;
- $f = \langle x_1 := f^1(X_1), \ldots, x_n := f^n(X_n) \rangle$ is the computation when τ is executed and $\forall i \in [1, n] : x_i \in X, \forall i \in [1, n] : X_i \subseteq X$.

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 $\boldsymbol{\mathcal{B}}$ 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.

Definition 3 (Semantics of Atomic Components). *The semantics of an atomic component $B = \langle P, L, T, X \rangle$ is an LTS $S_B = \langle Q_B, P_B, \rightarrow \rangle$. We denote by $\boldsymbol{X}$ the set of valuations of the variables X, we have:*

- *$Q_B = L \times \boldsymbol{X}$ is the set of states;*
- *$P_B = P \times \boldsymbol{X}$ is the set of labels;*
- $\rightarrow = \{ \langle \langle l, v \rangle, p(v_p), \langle l', v' \rangle \rangle \mid \exists \tau = \langle l, p[X_p], g_\tau, f_\tau, l' \rangle \in T : g_\tau(v) \wedge v' = f_\tau(v_p/v) \}$.

After executing an interaction, atomic components receive new data on their ports. For a given port p, its variables (X_p) are updated with the new values v_p. The transition $\langle l, v \rangle \xrightarrow{p(v_p)} \langle l', v' \rangle$ is possible in the LTS iff the component has a transition $\exists \tau = \langle \ell, p[X_p], g_\tau, f_\tau, \ell' \rangle \in T$ such that:

- The guard before receiving the port variables v_p evaluates to *true*: $g_\tau(v) = true$.
- The application of the computation function $f_\tau(v_p/v)$ yields v'.

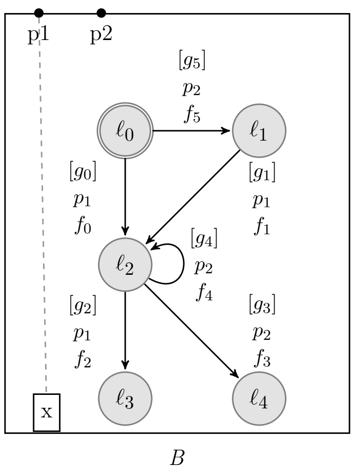


Figure 2.1: Example Atomic Component

Example 1 (Atomic Component). *Figure 2.1 depicts an atomic component B. B has two ports $\{p_1, p_2\}$ and one variable x. The port p_1 is attached to x. Additionally, B has a set of five locations $\ell_0, \ell_1, \ldots, \ell_4$ where ℓ_0 is the initial location. Each transition is represented by its guard followed by its port and then its computation. If B is in location ℓ_0 and that $g_5 = false$, but $g_0 = true$, then port p_1 is enabled. Possible enabled ports at ℓ_0 are shown in Table 2.1 . Assuming p_1 is selected to execute, and upon receiving data v through p_1, the variables are changed $x := v$ and then B will execute f_0 moving to ℓ_2.*

g_0	g_5	**Ports Enabled**
false	false	$\{\}$
true	false	$\{p_1\}$
false	true	$\{p_2\}$
true	true	$\{p_1, p_2\}$

Table 2.1: Possible enabled ports at ℓ_0

2.3 Interactions

Interactions serve as the glue that co-ordinates the components behavior. In BIP atomic components only expose their ports to the external system, as expected from components in component based design. Interactions are defined over ports. Combining the atomic components results in a composite component.

Definition 4 (Interaction). *An interaction a is a tuple $\langle P_a, F_a, G_a \rangle$, where:*

- *$P_a \subseteq \bigcup_{B \in \mathcal{B}}(B.ports)$ is a nonempty set of ports not containing more than one port per atomic component: $\forall B \in \mathcal{B} : |B.ports \cap P_a| \leq 1$. Let $X = \bigcup_{p_i \in P_a}(p_i.vars)$.*

- *F_a is a function to execute on the interaction, $F = \langle x_1 := f^1(X_1), \ldots, x_n := f^n(X_n) \rangle$ such that $\forall i \in [1, n] : x_i \in X, \forall i \in [1, n] : X_i \subseteq X$.*

- *G_a is a boolean expression, the guard expression on the interaction.*

For an interaction a, we denote P_a, G_a, F_a, as *a.ports*, *a.guard*, *a.func* respectively. We have additionally:

- The variables read during the interaction execution are defined by: $\mathrm{var}_r(a) = X_1 \cup X_2 \cup \ldots \cup X_n$;

- The variables read during the interaction execution are defined by: $\mathrm{var}_l(a) = \{x_1, x_2, \ldots, x_n\}$.

Definition 5 (Semantics of Composite Components). *Let $\mathcal{B} = \{B_1, \ldots, B_n\}$ be a set of atomic components with their associated LTS $S_{Bi} = \langle Q_{Bi}, P_{Bi}, \rightarrow \rangle$. Let γ be the set of interactions. The composite component determined by $\mathcal{B}$ and γ is $\mathcal{C} = \gamma(\mathcal{B})$ its semantics is the LTS $C = \langle Q, \gamma, \rightarrow \rangle$ where:*

- *$Q = Q_{B1} \times Q_{B2} \times \ldots \times Q_{Bn}$;*

- *$\rightarrow$ is the least set of transitions satisfying the following rule:*

$$\frac{\begin{array}{c} a = (\{p_i\}_{i \in I}, G_a, F_a) \in \gamma \qquad G_a(\{v_{pi}\}_{i \in I}) \\ \forall i \in I, q_i \xrightarrow{p_i(v_i)}_i q'_i \wedge v_i = F^i_a(\{v_{pi}\}_{i \in I}) \qquad \forall i \notin I, q_i = q'_i \end{array}}{\langle q_1, \ldots, q_n \rangle \xrightarrow{a} \langle q'_1, \ldots, q'_n \rangle}$$

  where v_{pi} is the valuation of the variables attached to port p_i and F^i_a is the partial function derived from F_a restricted to the variables associated with p_i.

An interaction a will be enabled iff its guard G_a is true and all of its ports are enabled. At a given time multiple interactions can be enabled but eventually one will be chosen to execute. Once chosen an interaction, a will execute its computation. The computation F_a is able to modify each of its port variables. The ports along with their respective valuations v_i will now execute, changing the associated atomic component appropriately.

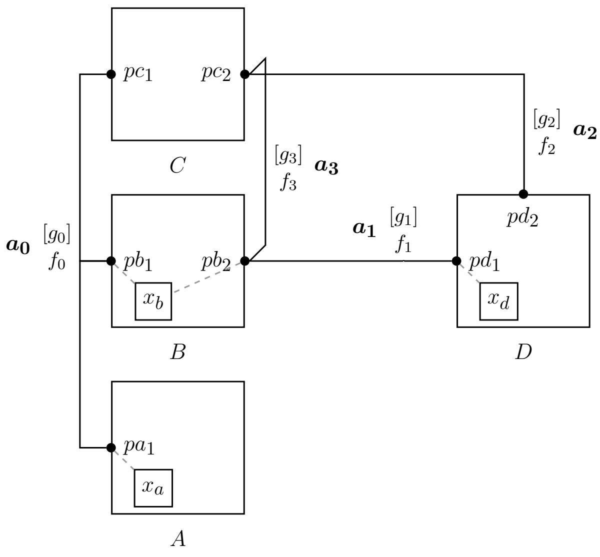


Figure 2.2: Composite Component

Example 2. *Figure Figure 2.2 depicts a composite component consisting of four atomic components $\mathcal{B} = \{A, B, C, D\}$ and four interactions $\gamma = \{a_0, a_1, a_2, a_3\}$. For interaction a_0 to be enabled, the ports $\{pa_1, pb_1, pc_1\}$ must be enabled and also g_0 must evaluate to true. When a_0 is executed, its computation f_0 handles assignment of port variables (data transfer) if necessary.*

2.4 Priorities

Multiple interactions can be enabled at once. Therefore, a strict partial ordering is defined on the interactions so as to choose one from the many. Formally:

Definition 6 (Priority). *A priority model π is a strict partial order on the set of interactions γ. Given a priority model π, we abbreviate $\langle a, a' \rangle \in \pi$ by $a \prec_\pi a'$. The LTS defining the behavior of a composite component $\mathcal{C} = \gamma(\mathcal{B})$ is $C = \langle Q, \gamma, \to \rangle$. Adding priority to $\mathcal{C}$ results in a new composite component $\mathcal{C}' = \pi(C)$, the associated semantics for $\mathcal{C}'$ is the LTS $C' = \langle Q, \gamma, \to_\pi \rangle$ where $\to_\pi$ is the least set of transitions satisfying the following rule:*

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

Ports Enabled	**Interactions Enabled**i	**Executed**
$\{pb_2, pc_2, pd_1\}$	$\{a_1, a_3\}$	a_3
$\{pa_2, pc_2, pd_1, pd_2\}$	$\{a_1, a_2, a_3\}$	a_2
$\{pa_1, pb_1, pc_1, pa_2, pc_2, pd_1, pd_2\}$	$\{a_0, a_1, a_2, a_3\}$	a_2

Table 2.2: Multiple Enabled Interactions

Example 3. *Table 2.2 lists the possible enabled interactions given a set of enabled ports. The priority layer helps us choose between these interactions. The last column depicts the chosen interaction to execute given the following order (least to highest priority):* a_0, a_1, a_3, a_2.

An execution step proceeds as follows:

1. For every atomic component, the guards of the outgoing transitions at the location are evaluated.

2. The transitions whose guards are true will have the ports enabled.

3. These enabled ports will then be used along with the interaction guards to determine which interactions are enabled.

4. The priority model then selects only one interaction.

5. Once selected, the interaction will execute its computation updating its port variables.

6. The ports will execute in their corresponding atomic components.

7. The associated atomic components will then execute their local computation functions associated with the transition labelled by the ports executing.

2.5 BIP System

Definition 7 (BIP System). *A BIP system S is a tuple $\langle C, Init, v\rangle$ where:*

- *C is the LTS of a composite component $\mathcal{C}$.*

- *$Init \in B_1.locs \times \ldots \times B_n.locs$ where $B_i \in \mathcal{B}$ is the set of initial locations of atomic components.*

- *$v \in \boldsymbol{X}^{Init}$ are the initial valuations of all atomic components variables $X^{Init} \subseteq \bigcup_{B\in\mathcal{B}}(B.vars)$.*

2.6 BIP Toolchain

The BIP tool-chain consists of a set of tools for modeling, execution and validation of BIP models [1, 11]. Figure 2.3 displays the relationships between those tools. Mainly, the tool-chain consists of the following.

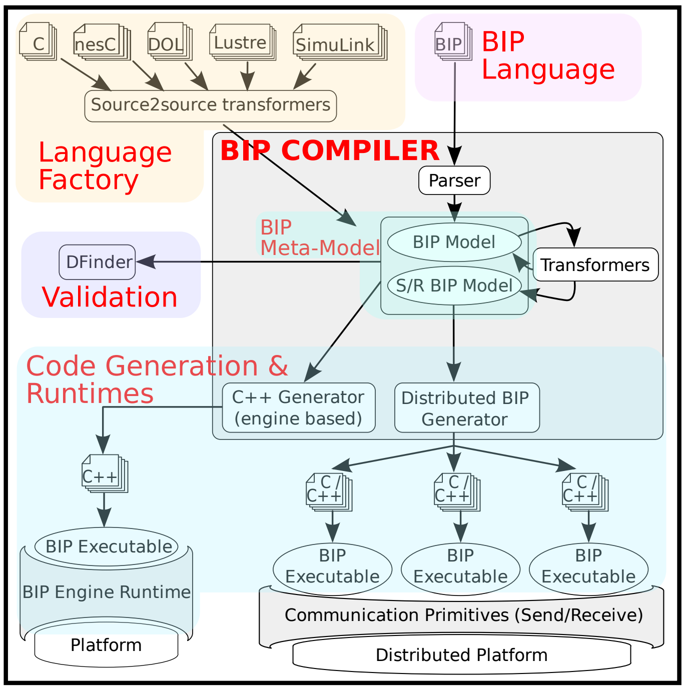


Figure 2.3: BIP Tool-Chain

Front-end. The front-end consists of an *editor* with a *compiler*, for describing textually a system in BIP language. The compiler generates a BIP model from BIP description source. The BIP model conforms to BIP meta-model which is built on top of the Eclipse Modeling Framework (EMF) [12].

Back-end. The back-end consists of several code generators that produce centralized and distributed implementations from a BIP model. For instance, it is possible to generate (1) a single-threaded C++ code running on the BIP engine; (2) multi-threaded C++ implementation; (3) distributed C++ implementation based on MPI or Socket.

Middle-end. The middle-end consists of source-to-source transformations that transform (1) a BIP model to another BIP model for optimization purposes; or (2) another language (e.g., C, Lustre, Simulink, etc.) to a BIP model.

Validation. The validation module includes DFinder [13] which is capable of compositional verification. For instance, it can be used to check invariants and deadlock-freedom of models. Moreover, using the validation module it is possible to do a statistical model checking of BIP models.

Chapter 3
ASPECT ORIENTED PROGRAMMING

Contents

3.1 Crosscutting Concerns 12

3.2 AOP Concepts 14

3.2.1 Joinpoints 14

3.2.2 Pointcuts 15

3.2.3 Advice 16

3.2.4 Aspect 17

3.2.5 Weaving 17

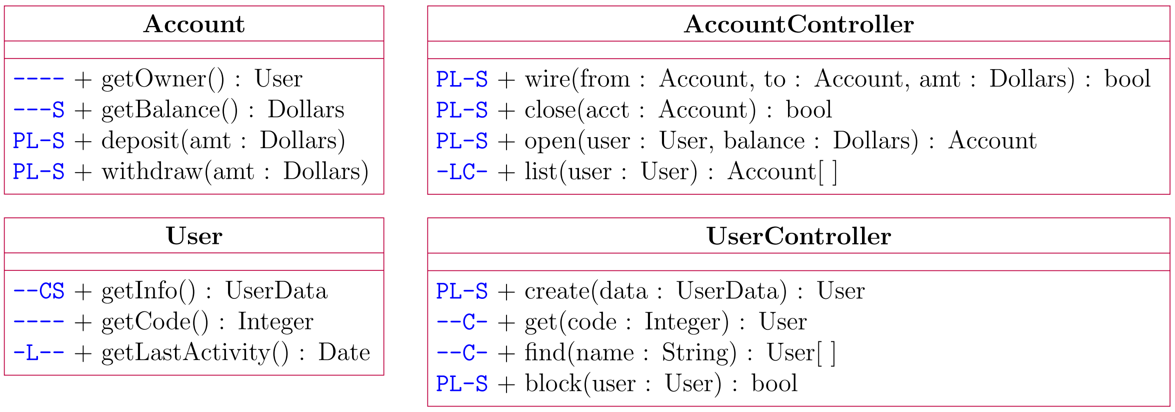
3.3 Overview of applying AOP in BIP 18

3.1 Crosscutting Concerns

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 found in the code. On the other hand, when logging is implemented, it is often scattered throughout multiple locations in the code. Logging and the main code are separate domains and represent different *concerns*. A concern is defined in [5] as a "*domain used as a decomposition criterion for a system or another domain with that concern*". Domains include logging, persistence and system policies like 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. The implementation of crosscutting concerns leads to two typical problems: *scattering* and *tangling* [9]:

- *Tangling* is the condition where concerns overlap in one region of the program. Consequently, enforcing one concern may affect others.
- *Scattering* can be seen as the dual of tangling. Scattering is the condition where one concern is spread across different regions of the program. Scattering

concerns violate encapsulation. Developers have to manually keep track the location of a specific concern in multiple areas of the system.



```
Account
---- + getOwner() : User
---S + getBalance() : Dollars
PL-S + deposit(amt : Dollars)
PL-S + withdraw(amt : Dollars)

AccountController
PL-S + wire(from : Account, to : Account, amt : Dollars) : bool
PL-S + close(acct : Account) : bool
PL-S + open(user : User, balance : Dollars) : Account
-LC- + list(user : User) : Account[ ]

User
--CS + getInfo() : UserData
---- + getCode() : Integer
-L-- + getLastActivity() : Date

UserController
PL-S + create(data : UserData) : User
--C- + get(code : Integer) : User
--C- + find(name : String) : User[ ]
PL-S + block(user : User) : bool
```

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

Figure 3.1: Multiple Concerns in a Simple System

Example 4 (Crosscutting Concerns). *Figure 3.1 illustrates four different concerns: logging, caching, persistence and policy. The policy given is a form of security policy. For example, the policy may (1) restrict access to bank account information based on customer information; (2) set a maximum withdrawal, wiring or deposit limit; (3) limit the number of accounts of users. 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,* `Account.withdraw` *method has three* tangled *concerns: persistence, logging and policy. Thus,* `withdraw` *method 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 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 [14] as "*a well modularized implementation of a crosscutting concern*". These concerns are separated from the main program logic and contained in separate logical units. In the example above four aspects would encapsulate each of the four concerns respectively: persistence, logging, caching and security policy. We will subsequently use AspectJ examples to illustrate the concepts. AspectJ [14] is an aspect-oriented extension to the Java programming language.

3.2 AOP Concepts

3.2.1 Joinpoints

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. Typically, a joinpoint can be seen as a node in the program's call graph where additional code can be hooked when the node is either entered or exited.

```
   public class AccountController {
2    private Policy securityPolicy;
     public boolean wire(Account from, Account to, Dollars amount) {
1      if(amount.isNegative()) {
       logger.warning("Negative debit");
       return false;
       }
       if(!securityPolicy.canWire(from, to, amount)) {
         logger.warning("Blocked by Policy");
         return false;
       }
       try {
3        Persistence.init();
         from.withdraw(amount);
         to.deposit(amount);
         Persistence.commit();
       } catch (Exception e) // Exception Handling
4        logger.warning(e);
         return false;
       }
       return true;
     }
   }
```

Listing 3.1: Joinpoints example in `AccountController.wire`

Example 5. *Listing 3.1 displays a sample code for* `AccountController.wire` *method. Joinpoints are dynamic execution points happening at runtime. Sample code is only used to describe execution steps. Below are some examples of joinpoints:*

1. *Capture a call to method* `isNegative` *in the object* `amount` *of class* `Dollars`*.*

2. *Capture access to private field* `securityPolicy` *in the object of class* `AccountController`*.*

3. *Capture access to a static method call* `init` *in the class* `Persistence`*.*

4. *Capture error handling.*

Note that, it is possible to capture a call to `Amount.isNegative` *within the call to* `AccountController.wire` *and targeting a specific instance of* `AccountController`*.*

To further understand the dynamic nature of the joinpoint, we consider three types of method selectors for AspectJ [3]. It provides method call, reception and execution. Method call captures the context of the calling object (or none for static invocation). Reception captures the receiving object upon receiving the call but prior to invoking the method. Finally, execution captures the receiving object's method invocation. Additional joinpoints include field access (`Set` and `Get`), constructors (call, reception, execution) and exception handling, etc.

3.2.2 Pointcuts

A *pointcut* refers to a set of *joinpoints* and execution context information. AspectJ provides several basic primitive to define pointcuts. Basic pointcuts can be composed and identified so as to increase re-usability. In AspectJ, pointcut designation can include *wildcard* expressions so as to easily match multiple methods. For instance, `call(packagenamevoid` $*$ `.set` $*$ $(*)$) matches all the setters that return `void` for all classes in `packagename` having any parameter signature. Pointcut regulates scattering by matching exactly the joinpoints needed to implement the concern.

Example 6 (Designating the Logging Joinpoints). *The user-defined pointcut in Listing 3.2 matches the execution points where the logging concern holds. Logging concerns are defined in Figure 3.1.* `call` *is used to designate the various method calls. The basic primitive* `call` *pointcuts are composed using the "or" (||) operator. The composed pointcut matches if any of the basic primitive* `call` *pointcuts match.*

```
pointcut log() :
  call(void Account.deposit(Dollars))       ||
  call(void Account.withdraw(Dollars))      ||
  call(* AccountController.*(*))            ||
  call(Date User.getLastActivity())         ||
  call(User UserController.create(UserData))||
  call(bool UserController.block(User)
```

Listing 3.2: Logging Pointcut

3.2.3 Advice

Advice defines the code to be executed at each specific joinpoint in a pointcut. Depending on the required behavior, multiple types of advice are defined so as to surround the joinpoint. AspectJ defines the following types: `before`, `after` and `around`. The code of an advise is a java method invocation. To account for exception handling, AspectJ provides two special cases of advice with type `after`: *after returning* and *after throwing*. All advices except for type `around` are strictly additive. That is, they cannot modify the original computation at the joinpoint. An advice of type `around` allows the user to bypass the computation at the joinpoint as it wraps the entry and exit of a joinpoint. The computation at the joinpoint is executed only if a special `proceed` instruction is called.

Example 7. *Listing 3.2 defines a pointcut that captures the various joinpoints of the logging concern. An advice that executes a computation after the joinpoints is shown in Listing 3.3*

```
after() returning() : log() {
  logger.log("Some Logging text");
}
```

Listing 3.3: A Simple Logging Advice

While the example illustrates running a simple logging message, it is often necessary to have access to more information. This information may contain parameters or return values. This is known as the joinpoint context. To do so, pointcut syntax in AspectJ allows defining parameters and binding them to the pointcut definition. Details on how this is possible is defined in [14]. For the sake of simplicity, the context is accessed with `thisJoinPoint` in the body's advice. `thisJoinPoint` object captures various context information including arguments and the method signature. The return value is captured using `returning`(`Typeid`).

```
after() returning(Object res) : log() {
  logger.info(thisJoinPoint.getSignature().toShortString()
    + Arrays.toString(thisJoinPoint.getArgs())
    + " -> " + res);
}
```

Listing 3.4: The Complete Logging Advice

Example 8 (Logging Advice). *Listing 3.4 displays a general purpose logging advice. To match any return type including* `void`*, we use the* `Object` *type as it captures all* *possible types. All parameters are captured by using the* `getArgs()` *method on the* `thisJoinPoint` *object.*

3.2.4 Aspect

Aspects serve as the modular unit that contains both advices and pointcuts. Moreover, aspects in AspectJ may contain their own methods and fields. This is referred 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). Inter-types allow for additional logic to handle complex concerns in one unit. Listing 3.5 displays the *Logging* aspect. It combines the pointcut, advice and a *Logger* object.

```
import java.util.logging.Logger;
public aspect Logging {
  private static Logger logger = Logger.getLogger(Logging.class.getName());

  pointcut log() :
    call(void Account.deposit(Dollars))       ||
    call(void Account.withdraw(Dollars))      ||
    call(* AccountController.*(*))            ||
    call(Date User.getLastActivity())         ||
    call(User UserController.create(UserData))||
    call(bool UserController.block(User)


  after() returning(Object res) : log() {
    logger.info(thisJoinPoint.getSignature().toShortString()
      + Arrays.toString(thisJoinPoint.getArgs())
      + " -> " + res);
  }
}
```

Listing 3.5: The Logging Aspect

3.2.5 Weaving

Concerns are isolated in their own modular unit, the *Aspect*. The main task of an Aspect Oriented Programming (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 *Aspect Weaving*. When multiple advices are woven onto the same joinpoint, the behavior of the advices is defined as follows:

1. `around` advice executes first, ordered from most specific to least specific. It must invoke `proceed` so as to allow the next `around` advice to execute. When all `around` advices have executed, control moves to step 2. Not calling `proceed` allows a user to stop the execution of the rest and even the existing code at the joinpoint.

2. `before` runs next from most specific to least specific.

3. At this stage the computation at the joinpoint is run. This runs the main logic.

4. Execution of *after returning* and *after throwing* is then run depending on the computation of the main code and the prior `afterreturning` advices. They match least specific first.

5. `after` advices are then run ordered by least specific first.

6. After all `after` are done running, the return value from the main computation (3) is returned to the inner most call of the `proceed` in the `around` (from 1) and that `around` keeps running.

7. When an `around` returns, control is passed to the surrounding `around` until all *around* advices are exhausted.

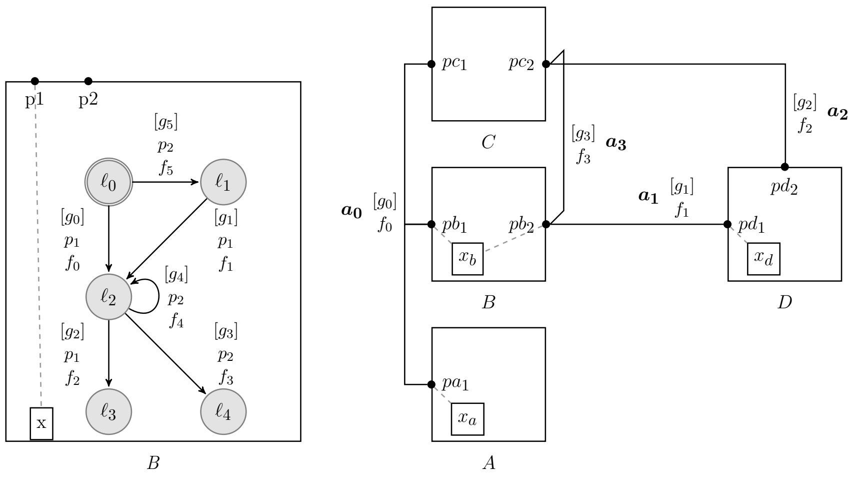
8. Control then returns to the end of the joinpoint.

Weaving can be done at compile-time or run-time. AspectJ tends to offset most weaving to be done at compile time while some constructs are needed still at run-time.

3.3 Overview of applying AOP in BIP

Developing component-based systems consist of progressively repeating the following two stages. The first stage consists of building atomic or basic components. This stage considers components as white-boxes. The second stage consists of composing components to build a complex one. This stage considers components as black-boxes.

It only uses the interfaces of components to compose them. BIP framework mimics exactly this development-flow. It starts first with developing atomic components, then compose them through their ports using interactions. Note that, this can be progressively applied to build composite components [15]. For this, we distinguish two views of aspects when dealing with BIP systems (see Figure 3.2).



(a) Local View (b) Global View

Figure 3.2: Local and Global Views

Local View. The local view (Figure 3.2(a)) deals with looking inside a component (i.e., first stage). This view is useful for the developers of atomic components in order refine their behaviors. Moreover, it allows to augment atomic components with many crosscutting concerns such as testing, runtime verification [16], enforcement and monitor synthesis. At the local view, concerns are at the level of location, state, transitions, guards and computations on transitions.

Global View. The global view (Figure 3.2(b)) deals when composing components (i.e., second stage). Thus, it considers interactions between components. In this case, components are considered as black-boxes. This view evaluates concerns that crosscut interactions. Concerns are at the level of the ports (interfaces), interactions and data transfer between components.

Recall that, the BIP framework is based on an abstract model with a well-defined operational semantics. This allows to easily map joinpoints, pointcuts, advices and aspects to the model and its semantics. The weaving procedure is then defined using model-to-model transformations along with their semantics.

Chapter 4
LOCAL ASPECTS

Contents

4.1 Preliminaries 21
4.2 Local Joinpoints 22
4.3 Local Pointcuts 24
4.3.1 Location 25
4.3.2 Variable Access 26
4.3.2.1 Variable Read Guard 26
4.3.2.2 Variable Read/Write Function 27
4.3.3 Ports 28
4.3.3.1 Port Execution 28
4.3.3.2 Port Enablement 29
4.3.4 Composing Pointcut Expressions 30
4.4 Local Advice 32
4.5 Local Aspect 33
4.6 Weaving Local Aspects 34
4.6.1 Strategy 34
4.6.2 Weaving $\langle CB, CA\rangle$ 35
4.6.3 Weaving $\langle PA, CB\rangle$ 36
4.6.4 Weaving $\langle ADD, CB\rangle$ 37
4.6.5 Weaving Reset Location Pairs 40
4.6.6 Weaving the Local Aspect 40

The first view deals with local aspects. Local aspects target atomic components in order to refine their behaviors. An atomic component's inner state is studied to locate possible points where crosscutting concerns happen.

4.1 Preliminaries

In this section, we introduce some preliminary concept and notations over BIP. Given an atomic component B, and a set of transitions $M \subseteq B.trans$, we define the following:

Definition 8 (Location Sets). *The origin set is defined as: $origin(M) = \{\tau.src \mid \tau \in M\}$. The destination set is defined as: $dest(M) = \{\tau.dest \mid \tau \in M\}$.*

The origin (resp. destination) consists of the source (resp. destination) locations of the transitions M.

Definition 9 (Relative Transitions). *The siblings set is defined as $siblings(M) = \{\tau \mid \tau.src \in origin(M) \wedge \tau \in B.trans\}$. The previous set is defined as $previous(M) = \{\tau \mid \tau.dest \in origin(M) \wedge \tau \in B.trans\}$*

The siblings (resp. previous) set consists of transitions that have their source (destination) locations within the origin locations of transitions M.

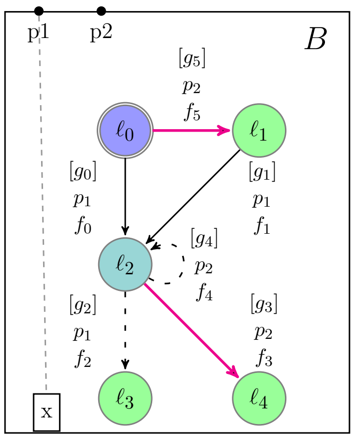


Figure 4.1: Origin, Destination, and Siblings

Example 9 (Origin, Destination, and Siblings). *Figure 4.1 shows an example of atomic component B with the set of transitions M colored in magenta. Location ℓ_2 is both in the origin and the destination set of M, this is because of the self loop. The destination set contains the locations to which the transitions in M lead to: $\{\ell_1, \ell_2, \ell_3, \ell_4\}$, and are colored in green. The origin set consists of the locations to which the transitions M are outbound: $\{\ell_0.\ell_2\}$, colored in blue. The transitions belonging to the siblings set are dashed.*

For a transition τ, below we define the variables evaluated in $\tau.guard$ and accessed (either read or write) in $\tau.func$.

Definition 10 (Accessed Variables). *Given a transition τ in component B, $X = B.vars$, and $\tau.func = \langle x_1 := f^1(X_1), \ldots, x_n := f^n(X_n)\rangle$ such that $\forall i \in [1, n] : x_i \in X, \forall i \in [1, n] : X_i \subseteq X$*

- varg(τ) *is the set of variables appearing in the guard of τ.*
- readvar(τ) $= X_1 \cup X_2 \cup \ldots \cup X_n$ *defines the set of variables with read-access (right-hand side) of the function of transition τ.*
- writevar(τ) $= \{x_1, x_2, \ldots, x_n\}$ *defines the set of variables with write-access (left-hand side) of the function of transition τ.*

4.2 Local Joinpoints

An atomic component has control locations, variables, ports and transitions labeled with guards and computation functions. At this level, concerns need to be managed at the following points: port execution/enablement, guard evaluation, access and modification of state's variables.

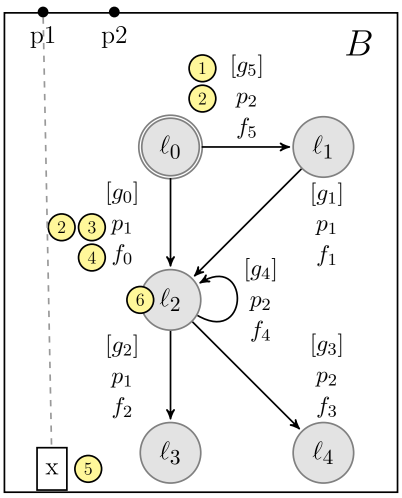


Figure 4.2: Identifying Joinpoints in Atomic Components

At the very base of an AOP design is to locate where concerns needs to be handled. The first step is to localize points where concerns happen. In an atomic component, concerns may be at the following points:

1. *Guard Evaluation*: designates the point where guards are evaluated.

2. *Port Enablement*: designates the point where given a port p in an atomic component at location ℓ one outgoing transition has its guard evaluated to true and is labelled by p.

3. *Port Execution*: once ports are enabled they are sent to the higher layer (i.e., the engine) to decide on the interaction to execute. After an interaction is executed, the ports that define the interaction are executed. This leads an atomic component to execute a transition labeled with its corresponding port.

4. *Local Computations*: designates the point where a certain transition's computation is executed. Concerns could be part of that computation.

5. *Variable Access*: designates read/write access to the state variables in a component.

6. *Location*: designates entry or exit of a given control location.

These various joinpoints are all represented as a transition τ in an atomic component and a range relative to that transition. This range maps the execution steps where a given joinpoint holds.

Definition 11 (Execution Point). *Given a transition τ. We identify four execution points:*

1. PA *(Previous After): designates the time after a local computation function is finished on any of the previous transitions that lead to τ;*

2. ADD *(Create): designates the need to add extra computation before evaluating or executing τ;*

3. CB *(Current Before): designates the time right before the current local computation on τ;*

4. CA *(Current After): designates the time right after the current local computation on τ.*

The set of execution points is $EP = \{PA, ADD, CB, CA\}$ and is strictly ordered, according to their executions: $PA \prec ADD \prec CB \prec CA$.

Figure 4.3 shows relative EPs for a selected transition τ. PA happens right after a previous transition has finished its computation. Once that computation is complete, the component enters the location ℓ_1. Upon entry to the location, ADD will happen and additionally the component. At this stage the atomic component evaluates its guards and determine enabled ports. An interaction is executed and eventually reaches the point where the atomic component must execute the computation f_1 of the transition τ. Before executing f_1 the component reaches the CB stage and after executing f_1 the component reaches the CA stage.

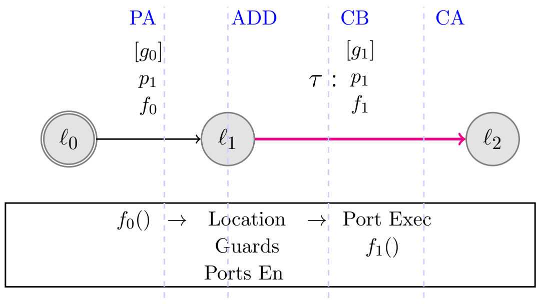


Figure 4.3: Execution Ordering in Atomic Components

Definition 12 (Local Joinpoint). *A local joinpoint in an atomic component B is represented as a tuple $\langle \tau, p_s, p_e \rangle$ where $\tau \in B.trans$ and $p_s, p_e \in EP$.*

p_s (resp. p_e) indicates the execution step relative to the start (resp. end) of the joinpoint. That is, the joinpoint starts right after p_s and ends right before p_e.

Example 10 (Example Joinpoint). *The joinpoint $\langle \tau, CB, CA \rangle$ in Figure 4.3 covers the execution frame ranging right at the start of the component executing f_1 and right at the end of executing f_1. This joinpoint captures a port execution.*

4.3 Local Pointcuts

Local pointcuts designate a group of joinpoints. We define local pointcut expressions to represent a combination of execution points. Then, we define $match_\ell(B, pc)$ that matches joinpoints given a local pointcut expression pc and an atomic component B.

Definition 13 (Local Pointcut Expression). *A pointcut expression $pc \in LPC$ describes a local pointcut expression. LPC is the set of local pointcut expressions. $LPC = 2^{atLocation(l)} \times 2^{readVarGuard(x)} \times 2^{readVarFunc(x)} \times 2^{write(x)} \times 2^{portEnabled(p)} \times 2^{portExecute(p)}$ where $x \in B.vars, l \in B.locs, p \in B.ports$.*

The operation $match_\ell(B, pc)$ yields a *pointcut match*. A *pointcut match* in an atomic component B is a tuple $\langle M, p_s, p_e \rangle$ where $M \subseteq B.trans$ and $p_s, p_e \in EP$. M is a set of transitions such that $\forall \tau \in M : \langle \tau, p_s, p_e \rangle$ is a joinpoint.

The operation $match_\ell$ is defined in the following sections. First, $match_\ell$ is defined for each of the primitive pointcut expressions: *atLocation*, *readVarGuard*, *readVarFunc*, *write*, *portEnabled*, *portExecute*. Then, we define the composition of primitive pointcut expressions.

4.3.1 Location

Given an atomic component B, the expression $atLocation(\ell)$ where $\ell \in B.locs$ captures B when its state is at location ℓ. This is matched: (1) just after the execution of transitions with destination location ℓ and; (2) right before transitions with source location ℓ that begin their computations. The frame of reference of $atLocation(\ell)$ consists of all transitions with source location ℓ.

Definition 14 (Matching Location). *Given an atomic component B, $match_\ell(B, atLocation(\ell)) = \langle M, PA, CB \rangle$ such that: $M = \{\tau \mid \tau \in B.trans \wedge \tau.src = \ell\}$.*

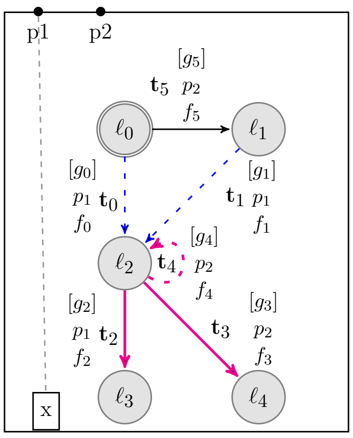


Figure 4.4: Matching $atLocation(\ell_2)$

Example 11. *Figure 4.4 shows $match_\ell(B, atLocation(\ell_2)) = \langle \{t_2, t_3, t_4\}, PA, CB \rangle$. The pointcut match contains the set of all outgoing transitions from ℓ_2, $M = \{t_2, t_3, t_4\}$. The incoming transitions to ℓ_2 are marked with dashes and are $M_{prev} = \{t_0, t_1, t_4\}$. Loops such as t_4 are both in the match, and the incoming transitions. PA designates the point after any transition in M_{prev} executes its computation.*

This marks the last step before entering ℓ_2 and therefore happens right before it. Upon exiting ℓ_2 the component must execute one of the computation functions of transitions in M. This marks the first step after exiting ℓ_2 and therefore happens right after it. Hence the execution frame of atLocation(ℓ_2) starts right after PA and ends right before CB.

4.3.2 Variable Access

Variable access happens in three different scenarios: (1) a variable is read when a guards is evaluated; (2) a variable is read during a transition function execution; (3) a variable is written to during a transition function execution.

Note that, we do not consider variable accesses that take place at interaction level in the local scope. This is considered in case of the global scope.

4.3.2.1 Variable Read Guard

First, we study variables read during guard evaluation. Guards are evaluated upon entry in a location and their evaluation ends right before a port is executed. Once guards are evaluated, enabled ports are evaluated. Then, atomic components wait to execute the selected ports by the engine, if any. This frame is identical to the *atLocation* pointcut, but instead can match multiple locations. Multiple locations are matched since transitions whose guards evaluate, a variable could be outbound from multiple locations.

Definition 15 (Matching Guard Variables). *Given an atomic component B, and a variable $x \in B.vars$, $match_{\ell}(B, readVarGuard(x)) = \langle M, PA, CB \rangle$ such that $M = \{siblings(\{\tau \mid \tau \in B.trans \wedge x \in \mathrm{varg}(\tau)\})\}$*

Transitions that have x in their guard expressions are selected first. The initial selection is referred to as M_{pre}. Guards are, however, evaluated upon entry to a given location and it terminates after exiting a location. Therefore effectively the match consists of all locations from which one or more transitions in M_{prev} are outbound. To indicate so, the selection is expanded to include $M = siblings(M_{pre})$. This is equivalent to matching *atLocation* at each of $\ell \in origin(M_{pre})$. As explained in the section before, entry in a location $\ell \in L$ happens right after PA, and exit happens right before CB.

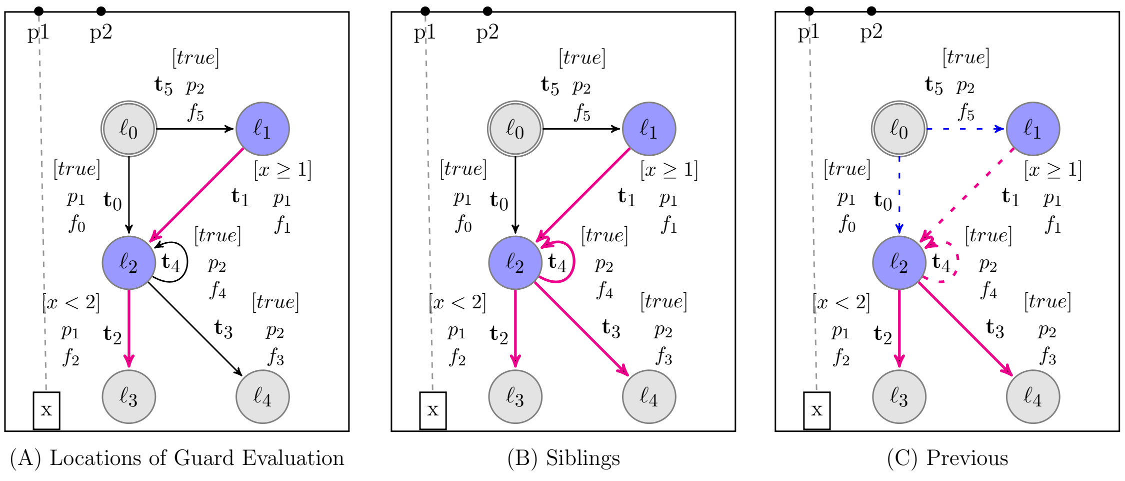


Figure 4.5: Matching $readVarGuard(x)$

Example 12. *Figure 4.5 shows* $match_{\ell}(B, readVarGuard(x)) = \langle \{t_1, t_2, t_3, t_4\}, PA, CB\rangle$. *The left-most image displays the selection of transitions which only contain x in their guards, $M_{pre} = \{t_1, t_2\}$. The locations highlighted in blue are the locations where the guard evaluation happens. These are $origin(\{t_1, t_2\}) = \{\ell_1, \ell_2\}$. The match is then expanded so as to match all transitions outbound from $\{\ell_1, \ell_2\}$. This results in selecting $M = siblings(M_{pre})$. It is possible for t_3 to excute, and upon execution of t_3 the component will leave ℓ_2. In that case it is still valid that x has been evaluated in the guards. The transitions that lead to our joinpoint are $previous(M) = \{t_0.t_1, t_4, t_5\}$. When a transition $\tau \in previous(M)$ finishes it computation the component will enter the joinpoint. If the component leaves ℓ_2 by executing t_4, $readVarGuard(x)$ will happen again.*

4.3.2.2 Variable Read/Write Function

Second, we consider read/write variables while executing a computation on a transition. These variables are accessed during the execution frame of a local computation. Therefore, they are bound by CB and CA for a given transition.

Definition 16 (Matching Function Variables). *Given an atomic component B and a variable $x \in B.vars$*[1]*:*

- $match_{\ell}(B, readVarFunc(x)) = \langle M_{read}, CB, CA\rangle$
  where: $M_{read} = \{\tau \mid \tau \in B.trans \wedge x \in \text{readvar}(\tau)\}$.

1. See Definition 10 for readvar and writevar

- $match_\ell(B, write(x)) = \langle M_{write}, CB, CA \rangle$
  where: $M_{write} = \{\tau \mid \tau \in B.trans \wedge x \in \text{writevar}(\tau)\}$.

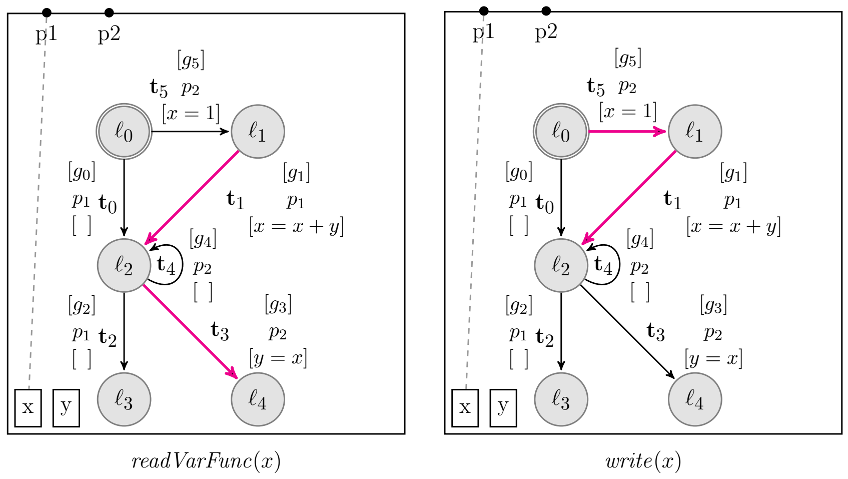


Figure 4.6: Matching Function Variables

Example 13. *Figure 4.6 expands the example to explicitly list the functions and add an additional variable y. The left image shows the transitions where x is read. The right image shows the transitions where x is written to. Variable access happens when a specific transition function is executed. The transition is executed right after its labelled port has been selected to execute. For example, upon execution of port p_1 at ℓ_1, t_1 is selected for execution. The component enters CB and then right after, the function $[x = x + y]$ will then execute, both reading and writing to x. Once the function is done executing the component reaches the point CA.*

4.3.3 Ports

Two pointcut expressions for ports are necessary: port execution and port enablement.

4.3.3.1 Port Execution

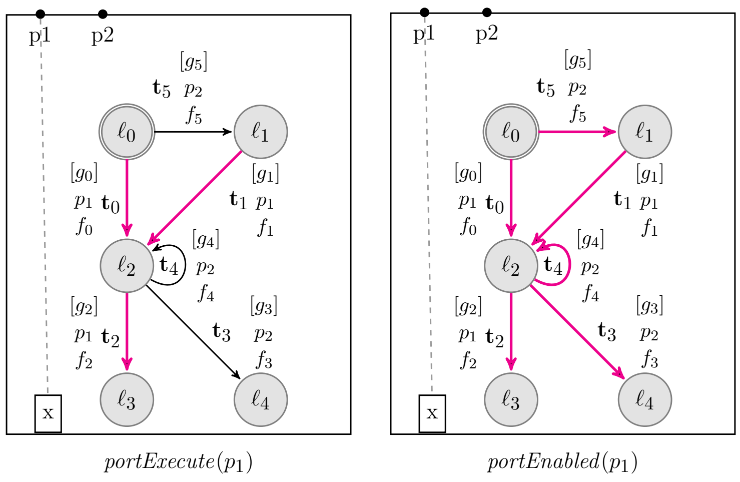
The execution of a port p, within an atomic component, corresponds to the firing of a transition labeled with port p, i.e., the firing of the following set of transitions $\{\tau.func \mid \tau \in B.trans \wedge \tau.port = p\}$. This pointcut selects transitions labeled with port p, and its execution frame starts after CB and ends before CA for transition labeled with port p.

Definition 17 (Matching Port Execution). *Given an atomic component B and a port $p \in B.ports$, $match_\ell(B, portEnabled(p)) = \langle M, CB, CA\rangle$ where $M = \{\tau \mid \tau \in B.trans \wedge \tau.port = p\}$.*

4.3.3.2 Port Enablement

Given an atomic component B with its corresponding semantics $\langle Q_B, P_B, \rightarrow\rangle$, a port p is enabled at state $q = \langle \ell, v\rangle \in Q_B$, where $v \in \mathbf{X}$, iff $\exists \tau \in\rightarrow$: $\big\langle \langle \ell, v\rangle, p(v_p), \langle \ell', v'\rangle \big\rangle$. This requires at least one outgoing transition from $\langle \ell, v\rangle$ to be labelled with port p. Since state information is accessible only at runtime, determining enabled ports requires to add extra transitions before all the transitions labeled with those ports. Pointcut of port enablement is assigned its own frame, *ADD*. This indicates that additional computation needs to be taken at the corresponding location, whose outgoing transitions labeled with ports to be matched. The port enablement joinpoint ends when a port is executed. This happens right after CB. It is possible at a given location to have multiple ports enabled. For this, the executed port may be be different from the port enabled. Consequently, it is important to match all transitions outbound from the location since they happens just after the port enablement.

Definition 18 (Matching Port Enablement). *Given an atomic component B and a port $p \in B.ports$, $match_\ell(B, portExecute(p)) = \langle M, ADD, CB\rangle$ such that $M = siblings(\{\tau \mid \tau \in B.trans \wedge \tau.port = p\})$.*



p1
p2
$[g_5]$ t_5 p_2 f_5
ℓ_0
ℓ_1
$[g_0]$ p_1 t_0 f_0
$[g_1]$ t_1 p_1 f_1
ℓ_2
t_4
$[g_4]$ p_2 f_4
$[g_2]$ p_1 t_2 f_2
$[g_3]$ t_3 p_2 f_3
x
ℓ_3
ℓ_4
$portExecute(p_1)$
$portEnabled(p_1)$

Figure 4.7: Matching Ports

Example 14. *Figure 4.7 on the left shows the transitions matched with $portExecute(p_1)$. To match the execution of port p_1 all transitions labeled with p_1*

are selected. These transitions are fired only if the port p_1 is executed. The Figure on the right shows the transitions matched with portEnabled(p_1). Note that, more transitions are selected since enablement of p_1 may be followed by the execution of port different than p_1 (e.g., p_2). Port p_1 is enabled at ℓ_2 iff g_2 is evaluated to true. That is, it is not possible to determine if p_1 is enabled without pre-evaluating g_2. Therefore, ADD is used even though port enablement (runtime/semantic dependent) is similar to guard evaluation (syntactic dependent). On the other hand, p_2 may be executed while p_1 is enabled at ℓ_2. Then, the component may fire either t_3 or t_4. Therefore, the joinpoint must include t_3 and t_4. However, if g_2 is evaluated to $false$, p_1 is not enabled. Thus the joinpoint must not include t_3 and t_4. These issues are handled during the weaving procedure.

4.3.4 Composing Pointcut Expressions

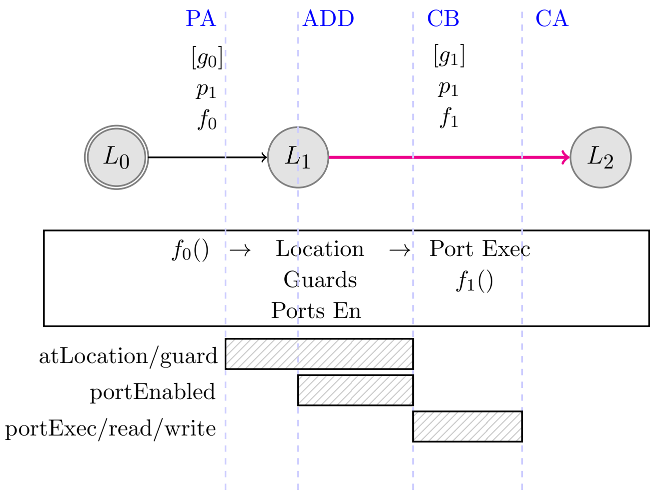


Figure 4.8: Pointcuts and Joinpoint Frames

A pointcut expression is composed of one or more of the primitive expressions: $atLocation(\ell), readVarGuard(x), readVarFunc(x), write(x), portEnabled(p)$, $portExecute(p)$. Figure 4.8 summarizes the execution time frames of their matching joinpoints.

Definition 19 (Combining Matches $\otimes$). *The combination of two pointcut matches $m_1 = \langle M_1, p_{s1}, p_{e1}\rangle$, $m_2 = \langle M_2, p_{s2}, p_{e2}\rangle$ is denoted by $r = m_1 \otimes m_2$ where r is a new pointcut match $r = \langle M_r, p_{sr}, p_{er}\rangle$ where:*

- *$M_r = M_1 \cap M_2$;*

- $p_{sr} = max(p_{s1}, p_{s2})$;
- $p_{er} = max(p_{e1}, p_{e2})$.

First, when combining matches, the result must ensure that the match is common. For this, the transitions from both matches are intersected to ensure that the result has transitions present in both. Second, the start of the execution frame denotes the execution point right before a match starts. A combination is considered to be started when a component has passed both p_{s_1} and p_{s_2} (i.e., the most delayed frame). Third, the end of the execution frame denotes the execution point right after the match has ended. A combination is considered to be ended when a component has passed both p_{e_1} and p_{e_2} (i.e., both frames have ended).

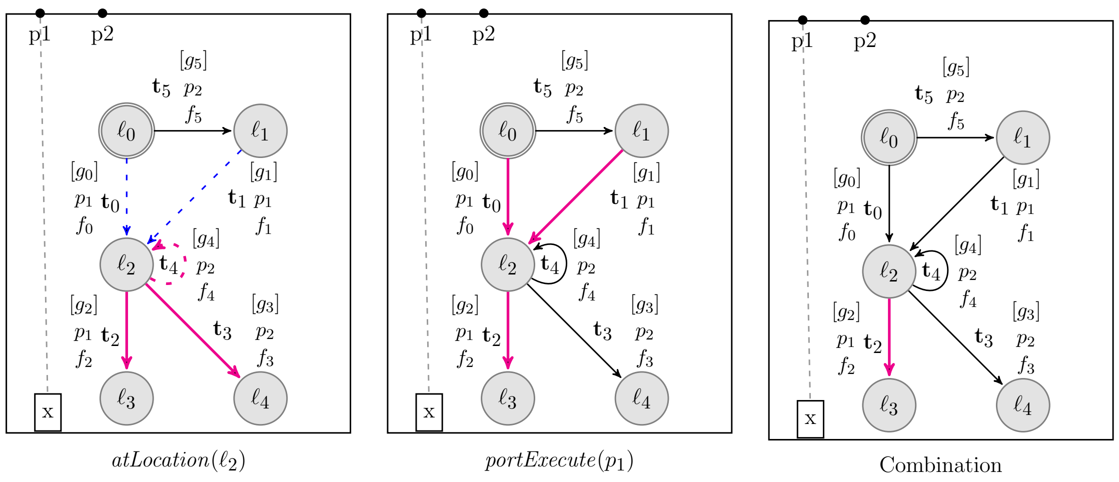


Figure 4.9: Combining Pointcut Matches

Example 15. *Figure 4.9 combines two pointcut matches* $r = m_1 \otimes m_2$, *where* $m_1 = match_\ell(B, atLocation(\ell_2))$, *and* $m_2 = match_\ell(B, portExecute(p1))$. *We have,* $m_1 = \langle \{t_2, t_3, t_4\}, PA, CB \rangle$, $m_2 = \langle \{t_0, t_1, t_2\}, CB, CA \rangle$, *and* $r = \langle \{t_2\}, CB, CA \rangle$. r *matches only transition* t_2. *The frame starts after both have started, i.e., directly after* CB. *This happens after port* p_1 *is selected for execution. If the frame were to start at* PA *then it would still be undetermined whether or not port* p_1 *would execute. The frame ends at* CA, *i.e., after the port* p_1 *is done executing. Right after this point, the component is neither at location* ℓ_2 *nor executing* p_1. *If the frame were to end at* CB, *the component would no longer be at location* ℓ_2 *but would still be executing* p_1. *The execution frame is bound to the execution of* f_2, *i.e., function of transition* t_2. *The result represents the execution of port* p_1 *on transitions with source location* ℓ_2.

Finally, several primitive pointcut expressions are pairwise matched ($\otimes$) according to the following definition. Note that, since $\cap$ and *max* are associative and commutative, $\otimes$ is also associative and commutative. That is, match order does not matter.

Definition 20 (Matching a Pointcut Expression). *Given an atomic component B, matching a pointcut expression $match_\ell(B, \{pc_1, pc_2, \ldots, pc_n\}) = match_\ell(B, pc_1) \otimes match_\ell(B, pc_2) \otimes \ldots \otimes match(B, pc_n)$.*

4.4 Local Advice

The advice determines the extra behavior to be added a joinpoint. A joinpoint is a tuple $\langle \tau, p_s, p_e \rangle$ consisting of a transition, start and end execution point. An advice is typically applied to a group of joinpoints. The pointcut match $\langle M, p_s, p_e \rangle$ designates a set of joinpoints. Given an atomic component B, a local advice may change its location, its variables $B.vars$, or additional extra variables V. An advice has then access to $X_{adv} = B.vars \cup V$. A local advice consists of three elements:

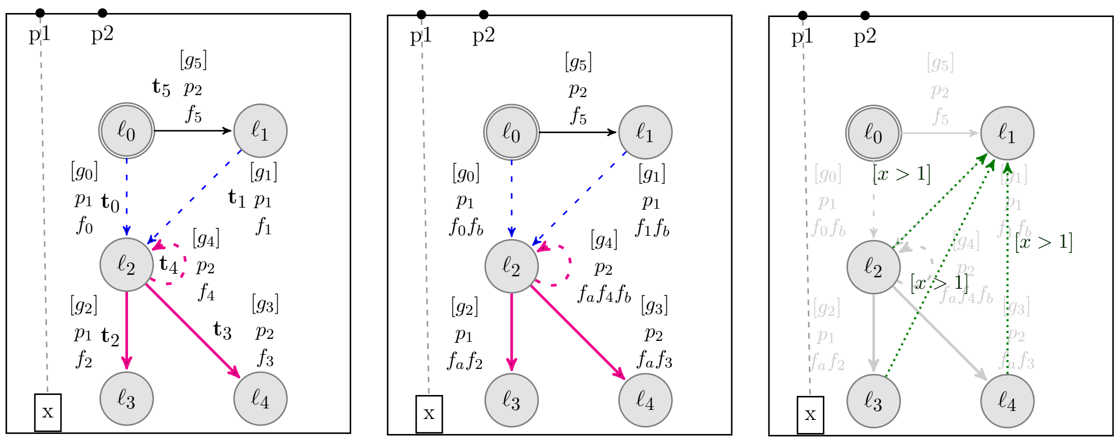
1. A before computation function $f_b = \left\langle x_1 := f^1(X_1), \ldots, x_n := f^n(X_n) \right\rangle$ such that $\forall i \in [1, n] : x_i \in X_{adv}, \forall i \in [1, n] : X_i \subseteq X_{adv}$. This computation happens right at p_s, the point that directly precedes the joinpoint.

2. An after computation function $f_a = \left\langle y_1 := f^1(Y_1), \ldots, y_m := f^n(Y_m) \right\rangle$ such that $\forall i \in [1, m] : y_i \in X_{adv}, \forall i \in [1, m] : Y_i \subseteq X_{adv}$. This computation happens right at p_e, the point that directly follows the joinpoint. Note that, the execution of f_b must be followed by the execution of f_a. Since the execution of f_a implies that a joinpoint was matched and its start point executed. Hence, the end point of that joinpoint must also execute.

3. A reset location set R. R is a set of tuples $r = \langle \ell, g \rangle$. Each tuple r indicates that after the end of the joinpoint the component has to modify its location to ℓ if guard g evaluates to *true*.

Note that f_b and f_a computations are not necessarily before and after τ, they could happen before or after a different transition depending on the execution frame of the joinpoint.

Example 16. *Figure 4.10 shows the advice $f_b = [x = x + 1]$, $f_a = [x = x - 1]$ and $R = \{\langle \ell_1, (x > 1) \rangle\}$ applied to $match(B, atLocation(\ell_2)) = \langle \{t_2, t_3, t_4\}, PA, CB \rangle$.*


p1
p2
ℓ_0
ℓ_1
ℓ_2
ℓ_3
ℓ_4
x
$[g_5]$ $\mathbf{t}_5$ p_2 f_5
$[g_0]$ p_1 $\mathbf{t}_0$ f_0
$[g_1]$ $\mathbf{t}_1$ p_1 f_1
$[g_4]$ $\mathbf{t}_4$ p_2 f_4
$[g_2]$ p_1 $\mathbf{t}_2$ f_2
$[g_3]$ $\mathbf{t}_3$ p_2 f_3
$f_0 f_b$
$f_1 f_b$
$f_a f_4 f_b$
$f_a f_2$
$f_a f_3$
$[x > 1]$

atLocation(ℓ_2) Applying f_b and f_a Applying R

Figure 4.10: Applying Advice to *atLocation*(ℓ_2)

The figure illustrates the behavior expected after the weaving is done. f_b executes at PA and f_a at CB. Thus, function f_b executes after f_0, f_1 and f_4, while function f_a executes before f_2, f_3 and f_4. After f_a is done executing, the component enters a location $\ell \in dest(M) = \{\ell_2, \ell_3, \ell_4\}$ depending on the transition executed. Then, R forces the component to move to ℓ_1 if $x > 1$ evaluates to true.

4.5 Local Aspect

A *Local Aspect* is defined on an atomic component as a combination of: (1) its extra variables; (2) a pointcut expression; (3) an advice. Formally:

Definition 21 (Local Aspect). *A local aspect LA is defined as a tuple $\langle B, pc, V, f_b, f_a, R\rangle$, let $X_{adv} = B.vars \cup V$ where:*

- *B is an atomic component;*
- *$pc \in LPC$ is a pointcut expression;*
- *V is a set of extra variables;*
- *$f_b = \langle x_1 := f^1(X_1), \ldots, x_n := f^n(X_n)\rangle$, where $\forall i \in [1, n] : x_i \in X_{adv}, \forall i \in [1, n] : X_i \subseteq X_{adv}$, is the computation to be executed before the pointcut match;*
- *$f_a = \langle x_1 := f^1(X_1), \ldots, x_m := f^n(X_m)\rangle$ such that $\forall i \in [1, m] : x_i \in X_{adv}, \forall i \in [1, m] : X_i \subseteq X_{adv}$ is the computation to execute after the pointcut match.*
- *R is a set of tuples $r = \langle \ell, g\rangle$ where:*

- *$r.\ell \subseteq B.locs$ is a reset location;*
- *$r.g = g(X_{adv})$ is a guard to reset to location $r.\ell$.*

4.6 Weaving Local Aspects

Weaving is the process of injecting concerns at a given joinpoint. The weaving procedure enforces the behavior of an aspect into a given BIP system. The advice is woven as part of the BIP model using source-to-source transformations. This approach provides flexibility as the output of the weaving is also a BIP model. Weaving is integrated within the BIP tool-chain.

Given a composite component $\mathcal{C} = \pi(\gamma(\mathcal{B}))$ weaving a local aspect onto an atomic component $B \in \mathcal{B}$ yields $\langle \mathcal{C}', m \rangle$. The result of the weave is a new composite component $\mathcal{C}'$ and a map function m. The resulting composite component is $\mathcal{C}' = \pi'(\gamma'((\mathcal{B} \setminus B) \cup B'))$ where B' is the instrumented atomic component. Hereafter, T and T' denote the transitions $B.trans$ and $B'.trans$ respectively. A transition is denoted by $t = \langle \ell, p, g, f, \ell' \rangle \in T$. $m : T \rightarrow 2^{T'}$ is an injective function that keeps track of the modified transitions. This is useful for weaving multiple aspects.

4.6.1 Strategy

Recall that, an advice may update components' variables at the before and after computations. However, when a joinpoint is matched, the execution of the before computation must ensure the execution of the after computation. For one aspect, this is guaranteed since when composing joinpoints we take the latest before and the earliest after (see Definition 19). Note that, this is, however, not ensured if the user's advice results 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. However, since the weaving transforms a BIP model into another one, it is particularly possible to use verification tools (such as $DFinder$) on the transformed model to check deadlocks. On the other hand, when weaving several aspects we need to add some extra variables to ensure the stipulated property.

The general weaving strategy uses one boolean variable per aspect b_{aop}. $b_{aop}$is set to $false$ right before matching the joinpoint and set to $true$ upon joinpoint entry. The corresponding computations to manipulate the boolean variable are $f_{set} = [b_{aop} := true]$ and $f_{clear} = [b_{aop} := false]$.

For a given execution frame $\langle p_s, p_e \rangle$ the before computation is woven at p_s while the after computation is woven at p_e. Table 4.1 lists the frames obtained from matching primitive pointcut expressions. When exhausting all combinations, the following frames are only possible: $\langle PA, CB \rangle, \langle CB, CA \rangle, \langle ADD, CB \rangle$.

Primitive Pointcut Expression	**Match Execution Frame** $\langle p_s, p_e \rangle$
$atLocation, readVarGuard$	$\langle PA, CB \rangle$
$readVarFunc, write, portExecute$	$\langle CB, CA \rangle$
$portEnabled$	$\langle ADD, CB \rangle$

Table 4.1: Summary of Pointcut Match Frames

Weaving of reset location is similar to all execution frames, it will be described last. The following advice for the frame weaving is used : $adv = \langle f_b, f_a, \{\} \rangle$. The advice adv provides both the before and after computation but no reset location.

4.6.2 Weaving $\langle CB, CA \rangle$

Given a pointcut match $\langle M, CB, CA \rangle$ weaving $\langle f_b, f_a, \{\} \rangle$ onto the match requires weaving f_b on CB and f_a on CA. Recall that CB indicates the execution point right before executing a computation f and CA indicates the execution point right after executing f, where f is the computation function of a transition in M. Therefore f_b (resp. f_a) is simply preceded (resp. succeeded) to f, and hence resulting in $f' = [f_b; f; f_a]$. All transitions leading to the transitions in M, namely $previous(M)$ must invoke f_{clear} after finishing their computation. After executing f_a, f_{set} is invoked to indicate that the joinpoint matched. This is used as a guide by the reset location to indicate that the joinpoint was matched.

The following three rules define the required weaving procedure corresponding to $\langle CB, CA \rangle$:

$$\frac{t \in M \quad p_s = CB \quad p_e = CA}{t' = \langle \ell, p, g, f_b f f_a f_{set}, \ell' \rangle \in T' \quad m(t) = \{t'\}} \qquad \langle CB, CA \rangle$$

$$\frac{t \in previous(M) \setminus M \quad p_e \neq PA}{t' = \langle \ell, p, g, f f_{clear}, \ell' \rangle \in T' \quad m(t) = \{t'\}} \qquad \langle Clear \rangle$$

$$\frac{t \in T \quad t \notin M \quad (t \notin previous(M) \vee p_s = ADD)}{t \in T' \qquad m(t) = \{t\}} \qquad \langle Copy \rangle$$

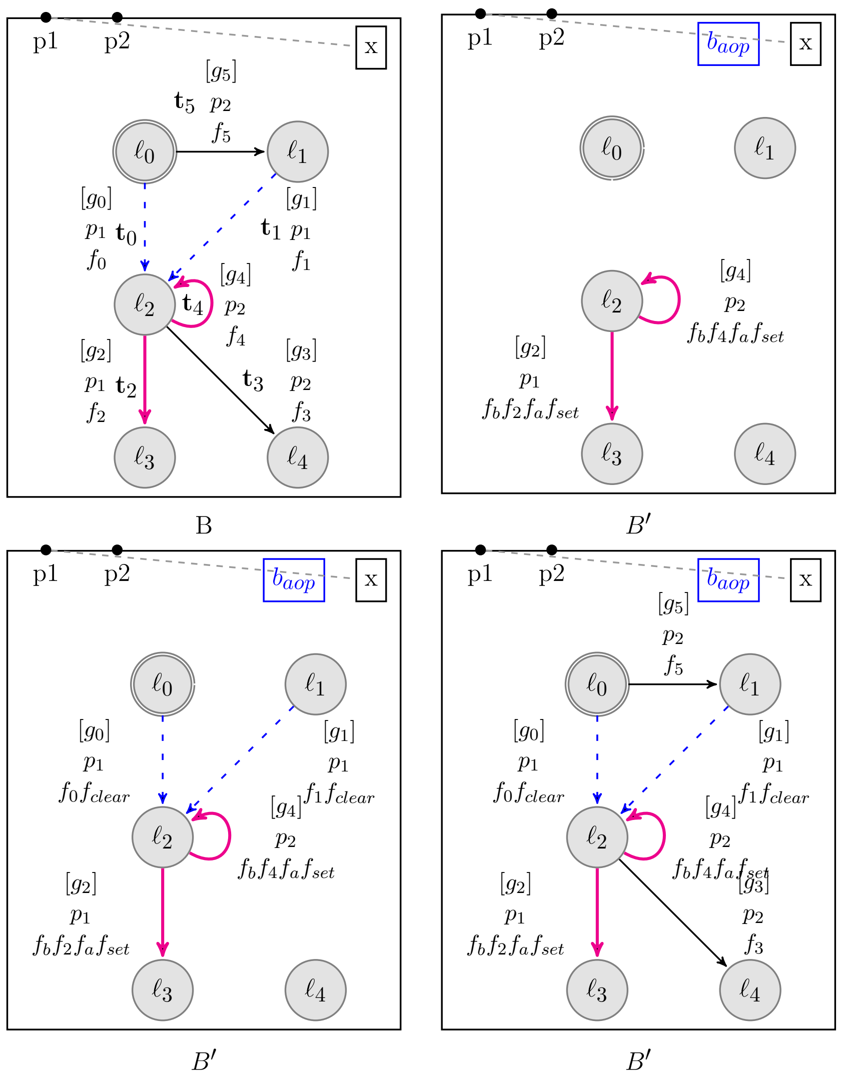

p1
p2
x
b_{aop}
$[g_5]$ $\mathbf{t}_5$ p_2 f_5
ℓ_0
ℓ_1
ℓ_2
ℓ_3
ℓ_4
$[g_0]$ p_1 $\mathbf{t}_0$ f_0
$[g_1]$ $\mathbf{t}_1$ p_1 f_1
$[g_4]$ $\mathbf{t}_4$ p_2 f_4
$[g_2]$ p_1 $\mathbf{t}_2$ f_2
$[g_3]$ $\mathbf{t}_3$ p_2 f_3
$f_b f_4 f_a f_{set}$
$f_b f_2 f_a f_{set}$
$f_0 f_{clear}$
$f_1 f_{clear}$
B
B'
B'
B'

Figure 4.11: Weaving $\langle CB, CA\rangle$

Example 17. *Figure 4.11 displays the weaving of adv onto* $\langle\{t_2, t_4\}, CB, CA\rangle$. *Using the rule* $\langle CB, CA\rangle$ *the transitions* t_2, t_4 *are added to* T' *while modifying their computation to account for the advice and to set the aop variable. Using the rule* $\langle Clear\rangle$ *the previous transitions* $previous(M) = \{t_0, t_1\}$ *have* f_{clear} *appended to their computation. Even though selected loop transitions (e.g.,* t_4*) do not invoke* f_{clear} *it is unnecessary to do so, as* f_{set} *is appended and therefore require no special rule. The last rule,* $\langle Copy\rangle$ *copies the unaffected transitions.*

4.6.3 **Weaving** $\langle PA, CB\rangle$

Given a pointcut match $\langle M, PA, CB\rangle$ weaving $\langle f_b, f_a, \{\}\rangle$ onto the match requires weaving f_b onto PA and f_a onto CB. Recall that PA indicates the point where the functions of transitions in $\mathit{previous}(M)$ have finished executing. CB indicates the point before the functions of transitions in M start executing. Therefore f_b is appended to all the functions of transitions in $\mathit{previous}(M)$ and f_a is appended to all the functions of transitions in M. Special care needs to be taken to loops since they could be both in M and $\mathit{previous}(M)$.

The following rules define the weaving procedure:

$$\frac{t \in M \setminus \mathit{previous}(M) \quad p_s = PA \quad p_e = CB}{t' = \langle \ell, p, g, f_a f, \ell' \rangle \in T' \quad m(t) = \{t'\}} \qquad \langle \mathit{Current} \rangle$$

$$\frac{t \in \mathit{previous}(M) \setminus M \quad p_s = PA}{t' = \langle \ell, p, g, f f_b f_{set}, \ell' \rangle \in T' \quad m(t) = \{t'\}} \qquad \langle \mathit{Previous} \rangle$$

$$\frac{t \in M \cap \mathit{previous}(M) \quad p_s = PA \quad p_e = CB}{t' = \langle \ell, p, g, f_a f f_b f_{set}, \ell' \rangle \in T' \quad m(t) = \{t'\}} \qquad \langle \mathit{Loop} \rangle$$

$$\frac{t \in T \quad t \notin M \quad (t \notin \mathit{previous}(M) \lor p_s = ADD)}{t \in T' \qquad m(t) = \{t\}} \qquad \langle \mathit{Copy} \rangle$$

Example 18. *Figure 4.12 displays the weaving of adv onto* $\mathit{match}(B, \mathit{atLocation}(\ell_2)) = \langle \{t_2, t_3, t_4\}, PA, CB \rangle$. *Using the rule* $\langle \mathit{Current} \rangle$, *the transitions* $\langle t_2, t_3 \rangle$ *are added to* T' *and contain the after* f_a *prepended to their function as it should happen* CB. *Using the rule* $\langle \mathit{Previous} \rangle$, *the transitions* $\langle t_0, t_1 \rangle$ *are added to* T' *and contain the before* f_b *appended to their function as it should happen* PA. *These transition would also set the aop variable indicating entry to the joinpoint. It is unecessary to clear upon entry as all entry paths will match, since executing* f_{clear} *followed by* f_{set} *results in* f_{set}. *Using the rule* $\langle \mathit{Loop} \rangle$ *loop transitions are copied correctly adding the* f_a *to execute before the function as the transition is previous and adding* f_b *followed by* f_{set} *as the transition is also current. The last rule,* $\langle \mathit{Copy} \rangle$ *copies the unaffected transitions.*

4.6.4 Weaving $\langle ADD, CB \rangle$

Given a pointcut match $\mathit{match}(B, pc) = \langle M, ADD, CB \rangle$ weaving $\langle f_b, f_a, \{\} \rangle$ onto the match requires extra considerations. The ADD 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 ADD. In the first step, we define the selected ports sp. Selected ports are the ports required to be matched as

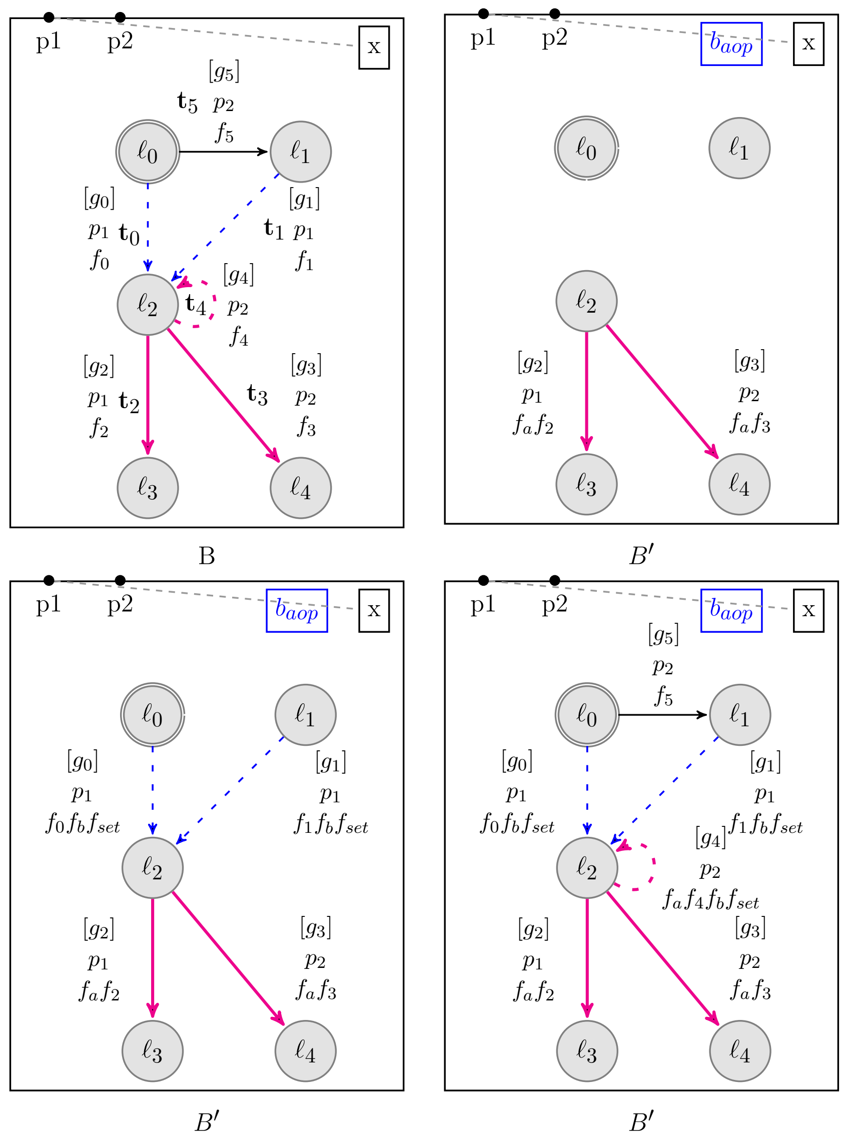

p1
p2
x
baop
ℓ_0
ℓ_1
ℓ_2
ℓ_3
ℓ_4
$[g_5]$ $\mathbf{t}_5$ p_2 f_5
$[g_0]$ p_1 $\mathbf{t}_0$ f_0
$[g_1]$ $\mathbf{t}_1$ p_1 f_1
$[g_4]$ $\mathbf{t}_4$ p_2 f_4
$[g_2]$ p_1 $\mathbf{t}_2$ f_2
$[g_3]$ $\mathbf{t}_3$ p_2 f_3
$f_a f_2$
$f_a f_3$
$f_0 f_b f_{set}$
$f_1 f_b f_{set}$
$f_a f_4 f_b f_{set}$
B
B'

Figure 4.12: Weaving $\langle PA, CB \rangle$

part of the pointcut.

$$SP = \{p \mid portEnabled(p) \in pc\}$$

Given a location ℓ a boolean operation $mkPortCondition(p, \ell, M)$, determines if a port p in a location ℓ is enabled [2]. Note that *portEnabled*, by Definition 18, matches all outgoing transitions from ℓ provided one of them is in M. This is due to *portEnabled* matching $siblings(M)$.

$$mkPortCondition(p, \ell, M) = \bigvee_{\tau \in M \wedge \tau.src = \ell \wedge \tau.port = p} (\tau.guard)$$

2. See page 29 for discussion on port enablement at a location.

If multiple ports SP are being matched for enablement at a location ℓ, then they must all be enabled at a location ℓ

$$mkAddGuard(SP, \ell, M) = \bigwedge_{p \in SP} (mkPortCondition(p, \ell, M))$$

Port enablement must be captured to handle ADD. To do so, given $\ell \in B.locs$ an extra control location $\ell^{\perp}$ is added to B'. Therefore, $B'.locs = L_{result} = B.locs \cup L_{temp}$

$$L_{temp} = \begin{cases} \left\{ loc^{\perp} \mid loc \in origin(M) \right\} & \text{iff } p_s = ADD \\ \emptyset & \text{otherwise} \end{cases}$$

ℓ and $\ell^{\perp}$ are then connected with two transitions. The first transition: (1) checks if the ports in SP are enabled, (2) executes f_b and, (2) does f_{set}. The second: (1) checks if the ports in SP are not enabled and, (2) invokes f_{clear}. A new port ip is created and added to $B'.ports = (B.ports \cup \{ip\})$ to label these transitions[3].

$$\frac{loc^{\perp} \in L_{temp} \quad p_s = ADD}{\left\langle loc, ip, mkAddGuard(SP, loc, M), f_b f_{set}, loc^{\perp} \right\rangle \in T' \quad \left\langle loc, ip, \neg mkAddGuard(SP, loc, M), f_{clear}, loc^{\perp} \right\rangle \in T'} \langle Create \rangle$$

Once ℓ' is created all its outgoing transitions from ℓ are redirected from ℓ to ℓ' and cloned. This results in two versions. The first checks for the joinpoint match ($b_{aop} = true$) and applies f_a. The second checks for ($b_{aop} = false$) and does not apply f_a weaving in CB.

$$\frac{t \in M \quad p_s = ADD}{t' = \left\langle \ell^{\perp}, p, b_{aop} \wedge g, f_a f, \ell' \right\rangle \in T' \quad t'' = \left\langle \ell^{\perp}, p, \neg b_{aop} \wedge g, f, \ell' \right\rangle \in T' \quad m(t) = \left\{ t', t'' \right\}} \langle Adjust \rangle$$

The unaffected transitions (i.e., $\notin M$) are then copied.

$$\frac{t \in T \quad t \notin M \quad (t \notin previous(M) \vee p_s = ADD)}{t \in T' \qquad m(t) = \{t\}} \langle Copy \rangle$$

Example 19. *Figure 4.13 displays weaving adv onto* $match(B, \{atLocation(\ell_2), portEnabled(p_1), portEnabled(p_2)\}) = \langle t_2, t_3, _t 4, ADD, CB \rangle$. *At first* $SP = \{p_1, p_2\}$ *is computed, then the origin is*

3. This simulates the behavior of an `if/else` construct without adding it to the computation functions.

determined, $origin(\{t_2, t_3, t_4\}) = \{\ell_2\}$. Once the origin is defined, all locations are copied, $L_{temp} = \left\{\ell_2^{\perp}\right\}$. The conditional expression for the joinpoint is then defined as $g_{add} = mkAddGuard(\{p_1, p_2\}, \ell_2, \{t_2, t_3, t_4\}) = g_2 \wedge (g_3 \vee g_4)$. 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. Using the rule $\langle Create\rangle$, two transitions are created per location upon entering the joinpoint. One transition executes iff the ports are both enabled, and another iff they are not both enabled. The first executes f_b from the advice and invoke f_{set}. The second simply clears invoking f_{clear}. The rule $\langle Adjust\rangle$ is then applied to copy over the originally outgoing transitions from ℓ_2, and creating two versions of them: one has f_a and guarded by b_{aop} and another executes normally and guarded by $\neg b_{aop}$. Remaining unaffected transitions are copied with the $\langle Copy\rangle$ rule.

4.6.5 Weaving Reset Location Pairs

Given am advice $\langle [\,], [\,], R\rangle$ applied to a pointcut match $\langle M, p_s, p_e\rangle$ reset location pairs are woven on all $dest(M)$. $\forall r \in R$ and $\forall \ell_{dest} \in dest(M)$. The following transitions are added to T'

$$\frac{r \in R \quad \ell_{dest} \in dest(M)}{\langle \ell_{dest}, ip, b_{aop} \wedge r.guard, f_{clear}, r.loc\rangle \in T'} \qquad \langle Reset\rangle$$

Moreover, the transitions are guarded by b_{aop} so as to only execute if the atomic component has entered the joinpoint. And once executed invoke f_{clear}.

Example 20. *Figure 4.14 shows the weaving procedure of $\langle [\,], [\,], \{\langle \ell_4, g_a\rangle, \langle \ell_1, g_b\rangle\}\rangle$ onto $\langle \{t_0, t_5\}, CB, CA\rangle$. The destination set is $dest(M) = \{\ell_1, \ell_2\}$. Reset locations are added to B'. In green we show the pair $r_1 = \langle \ell_4, g_1\rangle$ and in blue $r_2 = \langle \ell_1, g_b\rangle$.*

4.6.6 Weaving the Local Aspect

The rules presented for each frame are consolidated into one weave operation. The *local weave* applies an advice on joinpoints.

Definition 22 (Local Weave). *Given a composite component $\mathcal{C} = \pi(\gamma(\mathcal{B}))$, an atomic component $B = \langle P, L, T, X\rangle \in \mathcal{B}$, a local pointcut match $= \langle M, p_s, p_e\rangle$ such that $M \in B.trans$, and an advice $adv = \langle f_b, f_a, R\rangle$ to be applied on B with inter-type*

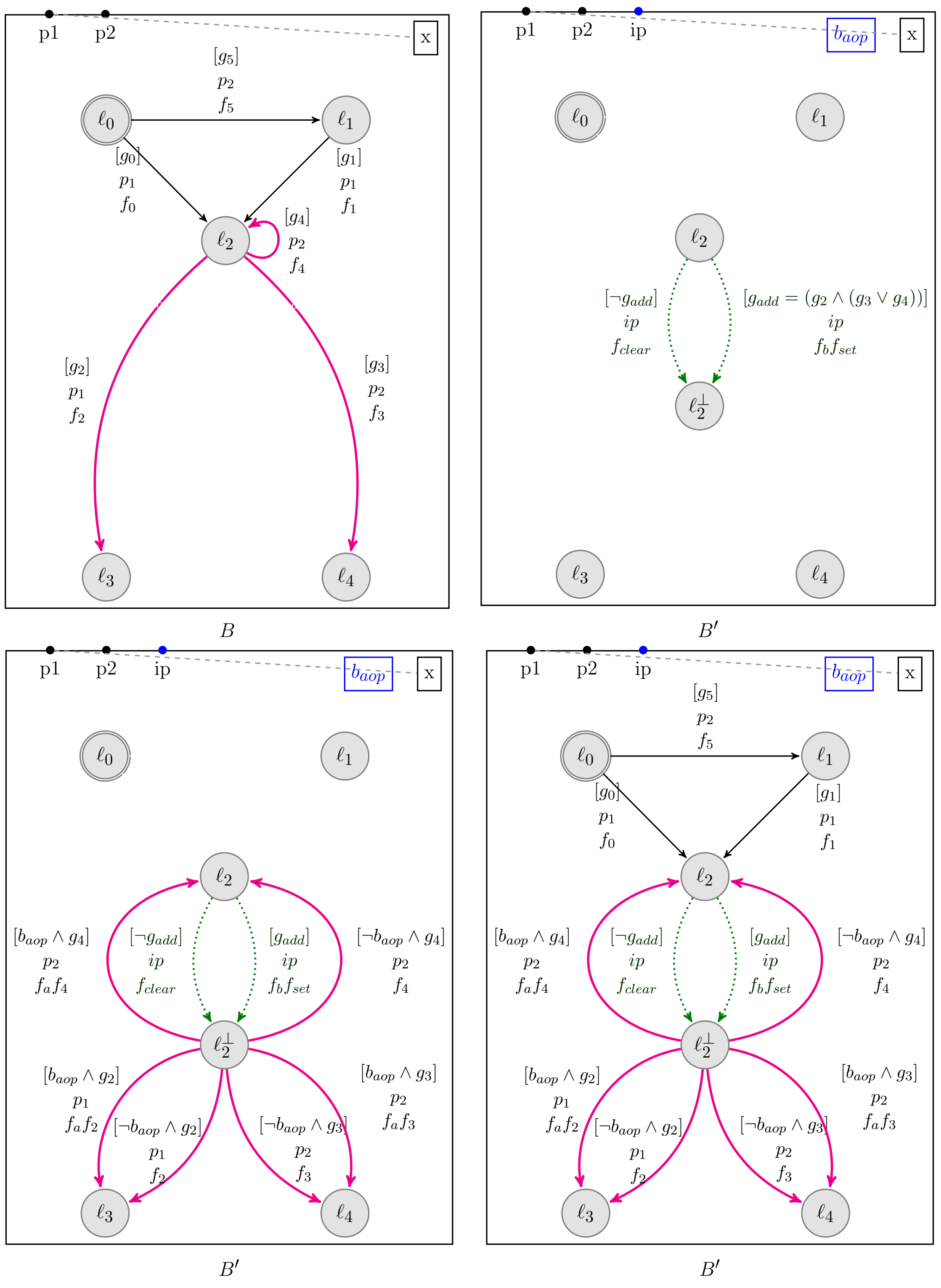

p1
p2
x
$[g_5]$ p_2 f_5
ℓ_0
ℓ_1
$[g_0]$ p_1 f_0
$[g_1]$ p_1 f_1
ℓ_2
$[g_4]$ p_2 f_4
$[g_2]$ p_1 f_2
$[g_3]$ p_2 f_3
ℓ_3
ℓ_4
B
p1
p2
ip
b_{aop}
x
ℓ_0
ℓ_1
ℓ_2
$[\neg g_{add}]$ ip f_{clear}
$[g_{add} = (g_2 \wedge (g_3 \vee g_4))]$ ip $f_b f_{set}$
$\ell_2^\perp$
ℓ_3
ℓ_4
B'
p1
p2
ip
b_{aop}
x
ℓ_0
ℓ_1
ℓ_2
$[b_{aop} \wedge g_4]$ p_2 $f_a f_4$
$[\neg g_{add}]$ ip f_{clear}
$[g_{add}]$ ip $f_b f_{set}$
$[\neg b_{aop} \wedge g_4]$ p_2 f_4
$\ell_2^\perp$
$[b_{aop} \wedge g_2]$ p_1 $f_a f_2$
$[\neg b_{aop} \wedge g_2]$ p_1 f_2
$[\neg b_{aop} \wedge g_3]$ p_2 f_3
$[b_{aop} \wedge g_3]$ p_2 $f_a f_3$
ℓ_3
ℓ_4
B'
p1
p2
ip
b_{aop}
x
$[g_5]$ p_2 f_5
ℓ_0
ℓ_1
$[g_0]$ p_1 f_0
$[g_1]$ p_1 f_1
ℓ_2
$[b_{aop} \wedge g_4]$ p_2 $f_a f_4$
$[\neg g_{add}]$ ip f_{clear}
$[g_{add}]$ ip $f_b f_{set}$
$[\neg b_{aop} \wedge g_4]$ p_2 f_4
$\ell_2^\perp$
$[b_{aop} \wedge g_2]$ p_1 $f_a f_2$
$[\neg b_{aop} \wedge g_2]$ p_1 f_2
$[\neg b_{aop} \wedge g_3]$ p_2 f_3
$[b_{aop} \wedge g_3]$ p_2 $f_a f_3$
ℓ_3
ℓ_4
B'

Figure 4.13: Weaving $\langle ADD, CB \rangle$

variables V; the local weave $weave_\ell(\mathcal{C}, B, match, V, adv)$ is equal to $\langle \mathcal{C}', m \rangle$. $\mathcal{C}' = \pi'(\gamma'((\mathcal{B} \setminus B) \cup B'))$ is the resulting composite component, where:

- *$B' = \langle P \cup \{ip\}, L \cup L_{temp}, X \cup V \cup \{b_{aop}\}, T' \rangle$ is the resulting atomic com-*

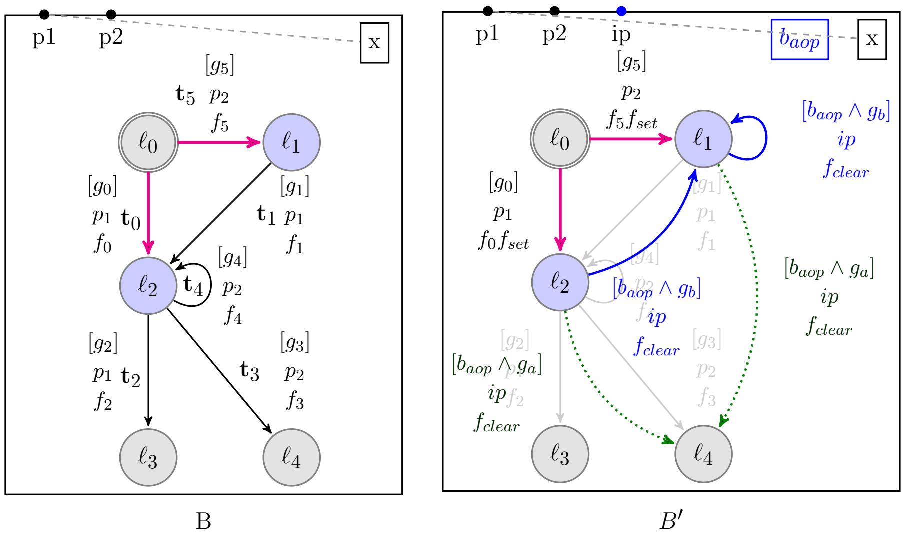


Figure 4.14: Weaving Reset Locations

ponent;

- $m : T \to 2^{T'}$ *is an injective function that keeps track of transitions;*
- b_{aop} *is a boolean variable created for each weave;*
- $L_{temp} = \begin{cases} \left\{loc^{\perp} \mid loc \in origin(M)\right\} & \textit{iff } p_s = ADD \\ \emptyset & \textit{otherwise} \end{cases}$.
- *Let* $t = \langle \ell, p, g, f, \ell' \rangle \in T$. T' *is the least set of transitions satisfying the following rules:*

$$\frac{t \in M \quad p_s = CB \quad p_e = CA}{t' = \langle \ell, p, g, f_b f f_a f_{set}, \ell' \rangle \in T' \quad m(t) = \{t'\}} \; \langle CB, CA \rangle$$

$$\frac{t \in M \setminus previous(M) \quad p_s = PA \quad p_e = CB}{t' = \langle \ell, p, g, f_a f, \ell' \rangle \in T' \quad m(t) = \{t'\}} \; \langle Current \rangle$$

$$\frac{t \in M \cap previous(M) \quad p_s = PA \quad p_e = CB}{t' = \langle \ell, p, g, f_a f f_b f_{set}, \ell' \rangle \in T' \quad m(t) = \{t'\}} \; \langle Loop \rangle$$

$$\frac{t \in previous(M) \setminus M \quad p_e \neq PA}{t' = \langle \ell, p, g, f f_{clear}, \ell' \rangle \in T' \quad m(t) = \{t'\}} \; \langle Clear \rangle$$

$$\frac{t \in previous(M) \setminus M \quad p_s = PA}{t' = \langle \ell, p, g, f f_b f_{set}, \ell' \rangle \in T' \quad m(t) = \{t'\}} \; \langle Previous \rangle$$

$$\frac{t \in M \quad p_s = ADD}{t' = \left\langle \ell^{\perp}, p, b_{aop} \wedge g, f_a f, \ell' \right\rangle \in T' \quad t'' = \left\langle \ell^{\perp}, p, \neg b_{aop} \wedge g, f, \ell' \right\rangle \in T' \quad m(t) = \{t', t''\}} \; \langle Adjust \rangle$$

$$\frac{loc^{\perp} \in L_{temp} \quad p_s = ADD}{\left\langle loc, ip, mkAddGuard(SP, loc, M), f_b f_{set}, loc^{\perp} \right\rangle \in T' \quad \left\langle loc, ip, \neg mkAddGuard(SP, loc, M), f_{clear}, loc^{\perp} \right\rangle \in T'} \; \langle Create \rangle$$

$$\frac{r \in R \quad \ell_{dest} \in dest(M)}{\langle \ell_{dest}, ip, b_{aop} \wedge r.guard, f_{clear}, r.loc \rangle \in T'} \; \langle Reset \rangle$$

$$\frac{t \in T \quad t \notin M \quad (t \notin previous(M) \vee p_s = ADD)}{t \in T' \qquad m(t) = \{t\}} \; \langle Copy \rangle$$

- $\gamma' = \gamma \cup \{a_{ip} = \langle ip, true, [\,] \rangle\}$;
- $\pi' = \begin{cases} \pi \cup \{\langle a, a_{ip} \rangle \mid a \in \pi\} & \textit{iff} \quad a_{ip} \notin \gamma \\ \pi & \textit{otherwise} \end{cases}$

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.

Both ip and a_{ip} are fixed for all possible weaves. Therefore, if $\mathcal{C}$ is the result of a previous local weave, it already contains them, in which case $ip \in B.ports$ and $a_{ip} \in \gamma$. In this case $B.ports = B'.ports$, $\gamma' = \gamma$ and the priorities remain unchanged $\pi' = \pi$, as no port and interaction has been effectively added.

It is possible for V, or a subset of $v \subseteq V$, to have already been woven if $\mathcal{C}$ is the result of an earlier weave. However since $B'.vars = B.vars \cup V$ they will not be woven again.

Now we are ready to define the weaving of a local aspect into a composite component.

Definition 23. Weaving a local aspect LA onto a composite component $\mathcal{C} = \pi(\gamma(\mathcal{B}))$ is defined as: $\mathcal{C}' = \mathcal{C} \lhd LA$ where:

- $LA = \langle B, pc, V, f_b, f_a, R \rangle$ *such that* $B \in \mathcal{B}$*;*
- $\mathcal{C}'$ *is the result from the local weave:* $\langle \mathcal{C}', m \rangle = weave_\ell(B, match_\ell(B, pc), V, \langle f_b, f_a, R \rangle)$.

Chapter 5
GLOBAL ASPECTS

Contents

5.1 Global Joinpoints 45
5.2 Global Pointcuts 46
5.3 Global Advice 48
5.4 Global Aspect 49
5.5 Weaving 49

In this chapter, we target the global view, i.e., interactions between atomic components (considered as black-boxes). Crosscutting concerns arise from coordination between components.

5.1 Global Joinpoints

At the global level, atomic components only export their ports, on which interactions are defined. Generally, each atomic component determines its enabled ports. Given the enabled ports and the guards of the interactions, the composite component executes one interaction which has: (1) all its ports enabled, (2) its guard evaluated to *true*, (3) there does not exist another interaction with higher priority which is also enabled. At the interaction level, the following operations exist: interaction enablement and interaction execution. For the scope of this thesis we only consider *interaction execution.* This is mainly due the complexity of matching *interaction enablement*, which requires to include the BIP engine as part of the BIP model. For this, it is better to handle it using a different approach by interfacing with the BIP engine. The *interaction execution* operation can express three joinpoints.

1. Synchronization between different atomic components
2. One or more atomic components sending data;
3. One or more atomic components receiving data.

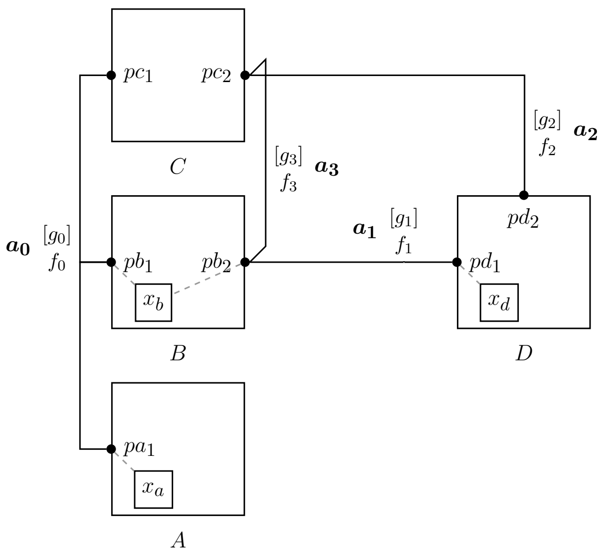


Figure 5.1: Example Composite Component

The joinpoint is modeled as an interaction execution and specifically, the execution of the interaction function.

Definition 24 (Global Joinpoint). *A* global joinpoint *in a composite component* $\gamma(\mathcal{B})$ *is defined as an interaction* $a \in \gamma$.

5.2 Global Pointcuts

In the global setting, joinpoints consist of interactions. Matching a group of interactions is done by matching their associated ports. For this, a global pointcut expression has two parts: the ports themselves and data transfer on those ports. Data transfer is captured by read or writes of the port variables in the computation function of the interaction.

Definition 25 (Global Pointcut). *A global pointcut expression is* $GPC = ports(\{p_1, \dots p_n\}) \times 2^{pcvars}$, *where* $pcvars = \{readPortsVars(\{x_1, \dots, x_m\}), writePortsVars(\{y_1, \dots, y_m\})\}$ *such that:*

- $\{p_1, \dots, p_n\} \subseteq \bigcup_{B \in \mathcal{B}}(B.ports)$;
- $\{x_1, \dots, x_m, y_1, \dots, y_m\} \subseteq \bigcup_{p \in \{p_1, \dots, p_n\}}(p.vars)$.

Recall that, the global joinpoint only consists of interaction execution. Thus, matching a global joinpoint yields to a set of interactions.

Definition 26 (Global Pointcut Match). *Given a composite component* $\mathcal{C} = \pi(\gamma(\mathcal{B}))$, $match_g(\mathcal{C}, pc) = \mathcal{I}$ *with* $pc \in GPC$ *and* $\mathcal{I} \subseteq \gamma$, *where:*

- $pc = ports(\{p_1, \dots p_n\}).readPortsVars(\{x_1, \dots, x_m\}).writePortsVars(\{y_1, \dots, y_m\})$;
- $\mathcal{I} = \{a \mid a \in \gamma \wedge \{p_1, \dots, p_n\} \subseteq a.ports \wedge \{x_1, \dots, x_m\} \subseteq \mathrm{var}_r(a) \wedge \{y_1, \dots, y_k\} \subseteq \mathrm{var}_l(a)\}$[1].

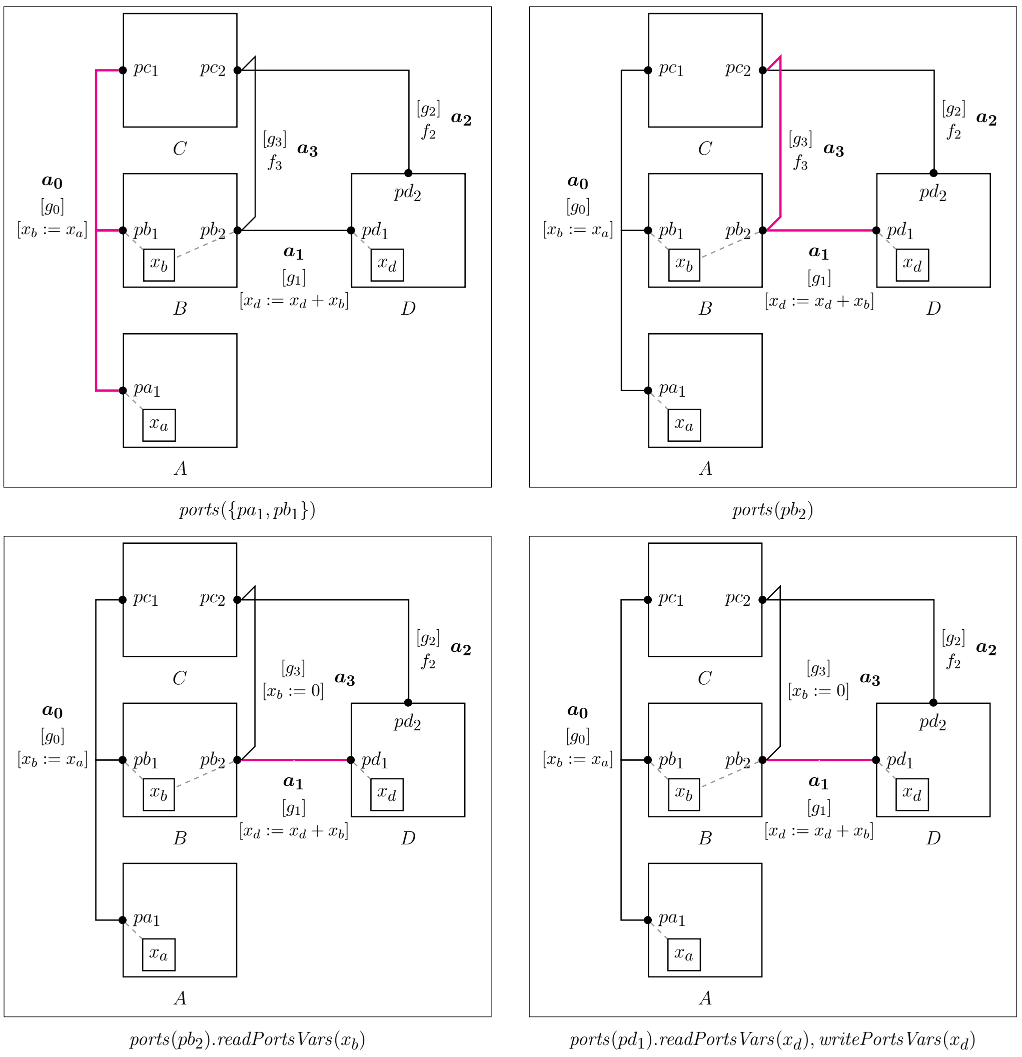


Figure 5.2: Matching Global Pointcuts

Example 21. Figure 5.2 shows the joinpoints obtained from matching four pointcuts:

1. *$ports(\{pa_1, pb_1\})$ matches all interactions that contain both $\{pa_1, pb_1\}$ in their ports. This will only match a_0 as it is the only interaction that has both ports.*
2. *$ports(\{pb_2\})$ matches all interactions that have pb_2 in their ports. The interactions $\{a_1, a_3\}$ will match since they are both connected to pb_2.*

1. See page 7 for var_r and var_l

3. *$ports(pb_2).readPortsVars(x_b)$ matches interactions that are connected to pb_2 and their computation must read the variable x_b associated with pb_2. This matches all interactions that depend on pb_2 sending data.*

4. *$ports(pd_1).readPortsVars(x_d), writePortsVars(x_d)$ matches interactions that are connected to pd_1 and their computation must read and write the variable x_d associated with pd_1. This matches all interactions that depend on pd_1 sending and receiving data.*

5.3 Global Advice

A global advice allows for extra computation to be executed at a global joinpoint. While the global pointcut match yields a set of interactions, a global advice has restricted access to the interaction's ports and their variables. Given a pointcut expression $pc = ports(\{p_1, \dots p_n\}).readPortsVars(\{x_1, \dots, x_m\}).writePortsVars(\{y_1, \dots, y_m\})$ and its match $\mathcal{I}$, the advice is restricted to the ports referenced by *ports* and extra variables V called the inter-type. The restriction ensures that an advice can only modify the ports that it matches, as interactions could include other ports. The non-matching ports are hidden from the advice[2]. Given an interaction a, the before (resp. after) advice is to execute before (resp. after) its computation function (i.e., $a.func$).

Definition 27 (Global Advice). *Given a set of ports P and inter-type variables V, $X_{adv} = V \cup \bigcup_{p \in P}(p.vars)$, are the advice variables. A global advice is a pair $\langle f_b, f_a \rangle$ such that:*

- *A before computation $f_b = \big\langle x_1 := f^1(X_1), \dots, x_n := f^n(X_n) \big\rangle$ such that $\forall i \in [1, n] : x_i \in X_{adv}, \forall i \in [1, n] : X_i \subseteq X_{adv}$;*
- *An after computation $f_b = \big\langle y_1 := f^1(Y_1), \dots, y_m := f^n(Y_m) \big\rangle$ such that $\forall i \in [1, m] : y_i \in X_{adv}, \forall i \in [1, m] : Y_i \subseteq X_{adv}$.*

2. As by application of the Law of Demeter [17]

5.4 Global Aspect

A global aspect binds a pointcut expression to its advice. It also adds additional inter-type variables. These variables are available to all interactions that match the pointcut. For instance, they can be used to keep track of global information, e.g., count the number of times an interaction is executed.

Definition 28 (Global Aspect). *A global aspect GA is a tuple* $\langle \mathcal{C}, V, pc, f_b, f_a \rangle$ *where:*

- $\mathcal{C}$ *denotes a composite component;*
- V *denotes the variables that are associated with the aspect;*
- $pc \in GPC$ *denotes the global pointcut expression, where* P *is the set of ports such that* $ports(P) \in pc$. $X_{adv} = V \cup \bigcup_{p \in P}(p.vars)$*;*
- f_b *denotes the advice's before computation over* X_{adv}*;*
- f_a *denotes the advice's after computation over* X_{adv}.

5.5 Weaving

The weaving of a global aspect requires weaving of the inter-type component and weaving of the advices. Weaving the inter-type component allows interactions to access inter-type variables V. The advice is woven to the interactions by attaching them to the inter-type component.

Weaving The Inter-type

The inter-type component is added to the system as a separate atomic component B_V. 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 *true*. This ensures that the port will not stop any other interaction from executing once connected to it.

Given a composite component $\mathcal{C} = \pi(\gamma(\mathcal{B}))$ and a set of variables V. Weaving the inter-type modifies $\mathcal{B}$ by generating $\mathcal{B}' = \mathcal{B} \cup B_V$ where $B_V = \langle \{p_V\}, \{\ell_0\}, \{\langle \ell_0, p_V, true, [\,], \ell_0 \rangle\}, V \rangle$ where $p_V = \langle p_V, V \rangle$. The component is identified by V. Therefore, for each V we can have at most one B_V associated with them.

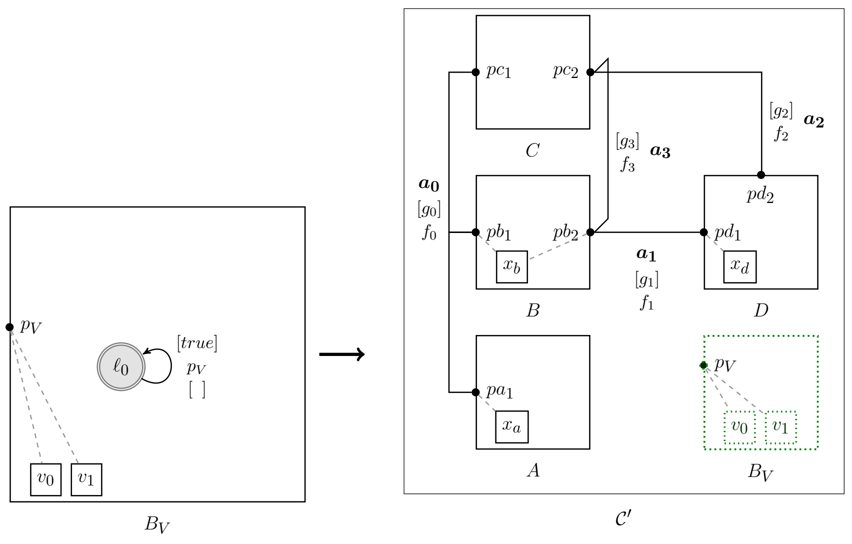


Figure 5.3: Inter-type Weaving

Example 22. *Figure 5.3 displays $\mathcal{C}' = \pi(\gamma(\mathcal{B} \cup \{B_V\}))$ where $V = \{v_0, v_1\}$ and $\mathcal{C} = \pi(\gamma(\mathcal{B}))$. A new atomic component is created B_V that has two local variables v_0 and v_1. And has its port p_V always enabled. The variables are attached to p_V.*

Weaving a Global Aspect

Once the inter-type component has been woven into the system, the advice is woven by connecting the existing interactions to it.

Definition 29 (Global Weave). *Given a composite component $\mathcal{C} = \pi(\gamma(\mathcal{B}))$, a global joinpoint $\mathcal{I}$, an inter-type V and a global advice $adv = \langle f_b, f_a \rangle$, the global weave is defined as $\langle \mathcal{C}', m \rangle = weave_g(\mathcal{C}, \mathcal{I}, V, adv)$ where:*

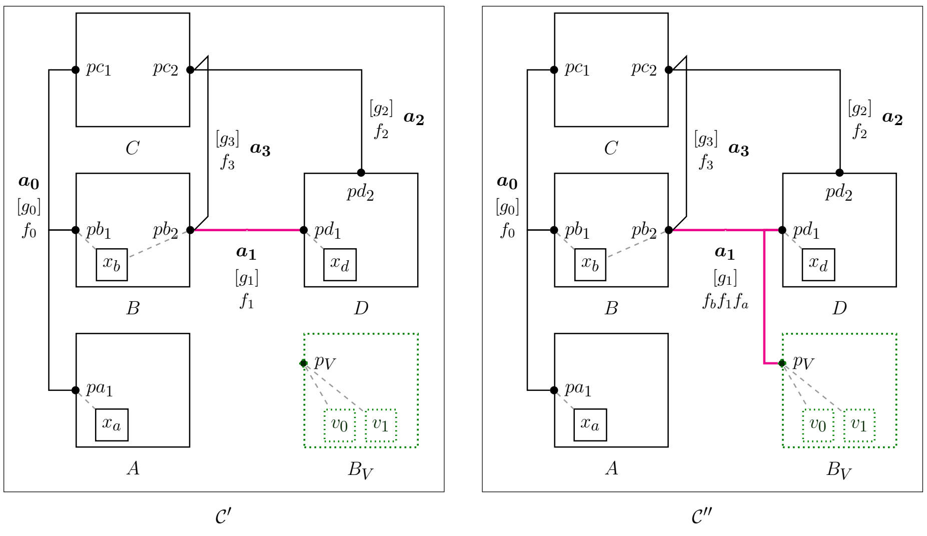
- *$\mathcal{C}' = \pi(\gamma'(\mathcal{B}'))$ is the new composite component;*
- *$m : \gamma \to \gamma'$ is a mapping that tracks changes to interactions;*
- *$\mathcal{B}' = \mathcal{B} \cup \{B_V\}$ is the new set of atomic components;*
- *$B_V = \langle \{p_V\}, \{\ell_0\}, \{\langle \ell_0, p_V, true, [\,], \ell_0 \rangle\}, V \rangle$ is the inter-type component identified by V;*
- *$p_V = \langle p_V, V \rangle$ is the port for the inter-type;*

- *γ' is the least set of interactions satisfying the following rules:*

$$\frac{a \in (\gamma \cap \mathcal{I})}{m(a) = \big\langle a.ports \cup p_{aop}, f_b\; a.func\; f_a, a.guard \big\rangle \in \gamma'} \qquad \langle Inject \rangle$$

$$\frac{a \in (\gamma \setminus \mathcal{I})}{a \in \gamma' \quad m(a) = a} \qquad \langle Default \rangle$$

The inter-type component B_V is added to $\mathcal{B}$. Since B_V is identified by V if V was already previously woven into the component then $B_V \in \mathcal{B}$ and $\mathcal{B}' = \mathcal{B}$. The interactions $\mathcal{I}$ are extended with the port p_V so as to have access to the inter-type and their computation function is prepended with f_b and f_a. The interactions priorities (π) are not modified, thereby preserving the priorities on the interactions.



pc_1
pc_2
C
$[g_2]$ f_2 $\boldsymbol{a_2}$
$[g_3]$ f_3 $\boldsymbol{a_3}$
$\boldsymbol{a_0}$ $[g_0]$ f_0
pd_2
pb_1
pb_2
pd_1
x_b
x_d
$\boldsymbol{a_1}$ $[g_1]$ f_1
B
D
p_V
pa_1
x_a
v_0
v_1
A
B_V
$\mathcal{C}'$
pc_1
pc_2
C
$[g_2]$ f_2 $\boldsymbol{a_2}$
$[g_3]$ f_3 $\boldsymbol{a_3}$
$\boldsymbol{a_0}$ $[g_0]$ f_0
pd_2
pb_1
pb_2
pd_1
x_b
x_d
$\boldsymbol{a_1}$ $[g_1]$ $f_b f_1 f_a$
B
D
p_V
pa_1
x_a
v_0
v_1
A
B_V
$\mathcal{C}''$

Figure 5.4: Completing The Weave

Example 23. *Figure 5.4 displays weaving the advice to $\mathcal{I} = match_g(C, ports(\{pb_2, pd_1\})) = \{a_1\}$.*

- *The interaction a_1 is connected to p_{aop} so as to allow access to $\{v_0, v_1\}$ on which f_b and f_a can operate.*

- *The computation 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, the interaction a_1 will be enabled when pb_2 and pd_1 are both enabled and g_1 is true. The extension to p_V does not affect enablement.*

- *Once a_1 is executed if f_b or f_a write onto $p_V.vars$ they will then be received in B_V and changed accordingly.*

We are now ready to define the weaving of a single global aspect GA.

Definition 30. *Weaving a global aspect GA onto a composite component $\mathcal{C} = \pi(\gamma(\mathcal{B}))$ is defined as: $\mathcal{C}' = \mathcal{C} \lhd_g GA$ where:*

- $GA = \langle \mathcal{C}, V, pc, f_b, f_a \rangle$;
- *$\mathcal{C}'$ is the result from the global weave: $\langle \mathcal{C}', m \rangle = weave_g(\mathcal{C}, match_g(\mathcal{C}, pc), V, \langle f_b, f_a \rangle)$.*

Chapter 6
ENCAPSULATING ASPECTS

Contents

6.1 Interference 53

6.2 Containers 54

6.3 Weaving Procedures 55

An aspect is the single association of a pointcut expression to a joinpoint. It may also include some extra variables. However, when weaving more than one aspect, specific problems and extra considerations need to be taken into account. This section deals with explaining how to weave multiple aspects, and elaborating on ways to group them.

6.1 Interference

Recall that multiple concerns may happen at one joinpoint in a program. Typically this can be seen as the *tangling* phenomenon. When secondary code is added to the joinpoint, it is possible to interfere with the existing code at the joinpoint. This behavior is called *interference*. Since not all concerns are independent, interference is an important issue to study.

Example 24. *Figure 6.1 displays the multiple weaving scenarios of two aspects asp_1 and asp_2 where:*

- $asp_1 = \langle B, pc, \{\}, f_b, f_a, \{\}\rangle$
- $asp_2 = \langle B, pc, \{\}, f'_b, f'_a, \{\}\rangle$

Both aspects share the same pointcut $pc = \{atLocation(\ell_1), portExecute(p_1)\}$. Let t_1 be the transition guarded by g_1 labeled by p_1 and whose computation is f_1. Both aspects will share the same match: $m = \langle\{t_1\}, CB, CA\rangle$. According to the execution frame both f_b and f'_b must be prepended to f_1. f_a and f'_a must be appended to f_1. The four scenarios display the possible arrangements to order f_b, f'_b, f_a and f'_a. The order would not be an issue if all those functions do not write to the component variables. For example in the case of logging, the ordering would not *matter. However suppose* $f_b = [x := 3]$ *and* $f'_b = [x := 2]$*. The variable* x *will be changed to either* 3 *or* 2 *by the end of the before section. This causes the original computation* f_1 *to behave non-deterministically if it depends on* x*.*

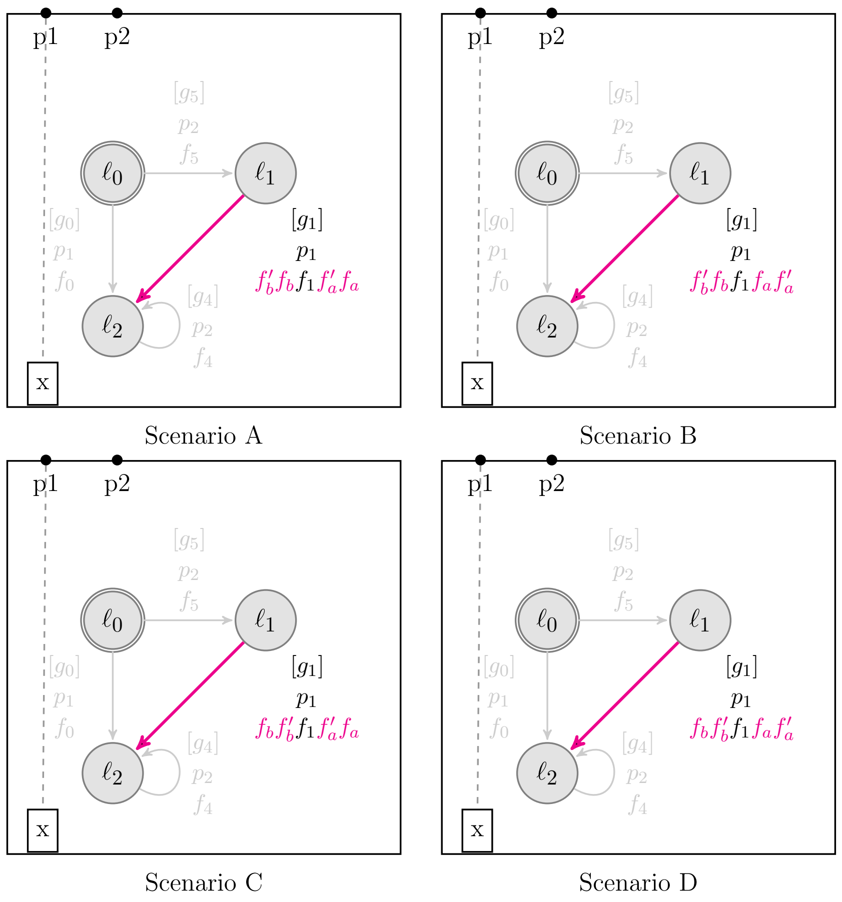


Figure 6.1: Interference In Atomic Components

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

6.2 Containers

Aspect containers encapsulate a group of aspects. Containers are the equivalent of an Aspect in AspectJ. A container consists of a sequence of aspects with shared properties. Local containers apply to local aspects. Global containers apply to global aspects.

Definition 31 (Local Container). *A local container is a tuple* $\langle \langle LA_1, \ldots, LA_n \rangle, B, V \rangle$ *such that* $\forall LA_i \in \{LA_1, \ldots, LA_n\} : LA_i$ *is applied to an atomic component* B *and* *has the inter-type V.*

Definition 32 (Glbbal Container). *A global container is a tuple $\langle\langle GA_1, \dots, GA_n\rangle, V'\rangle$ where $\forall GA_j \in \{GA_1, \dots, GA_m\} : GA_j$ has the inter-type V'.*

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

6.3 Weaving Procedures

To deal with interference we propose two approaches to compose aspects. These two approaches specify the system on which aspects match their pointcuts. The aspects are presented in a sequence $\langle asp_1, \dots, asp_n\rangle$ where $asp_1, \dots, asp_n$ are all either global or local aspects.

For a local aspect $LA = \langle B, pc, V, f_b, f_a, R\rangle$ we denote $B, pc, V, \langle f_b, f_a, R\rangle$ as $LA.B$, $LA.pc$, $LA.V$ and $LA.adv$, respectively. For a global aspect $GA = \langle \mathcal{C}, pc', V', f'_b, f'_a\rangle$ we denote $\mathcal{C}, pc', V', \langle f'_b, f'_a\rangle$ as $GA.\mathcal{C}$, $GA.pc$, $GA.V$ and $GA.adv$ respectively.

Weave Serial

The first compositional approach *weaveSerial* allows the pointcut of aspect asp_i where $i \in \{2, \dots, n\}$ to match any changes introduced by all aspects prior to it: asp_j where $\forall j : j < i$.

Definition 33 (*weaveSerial*). *Depending on the aspect type we have two operations:*

- *The procedure $weaveSerial_\ell$ applied to a sequence of local aspects $\langle asp_1, \dots, asp_n\rangle$ is defined as: $\mathcal{C}' = weaveSerial_\ell(\mathcal{C}, \langle asp_1, \dots, asp_n\rangle) = ((((\mathcal{C} \triangleleft asp_1) \triangleleft asp_2) \triangleleft ..) \triangleleft asp_n)$*

- *The procedure $weaveSerial_g$ applied to a sequence of global aspects $\langle asp_1, \dots, asp_m\rangle$ is defined as: $\mathcal{C}' = weaveSerial_g(\mathcal{C}, \langle asp_1, \dots, asp_m\rangle) = ((((\mathcal{C} \triangleleft_g asp_1) \triangleleft_g asp_2) \triangleleft_g ..) \triangleleft_g asp_m)$*

Weave All

The second compositional approach *weaveAll* allows the pointcut of aspect asp_i in the sequence to not match extra behavior of the advices of all prior aspects asp_j where $(\forall j : j < i)$. The non-matched behavior is the added transitions from the reset locations of local aspects. Therefore *weaveAll* only applies to local aspects, as global advices do not introduce new interactions.

The matching of pointcuts is separate from the weaving procedure, therefore it is possible to match the joinpoints of all the sequence of aspects before we start weaving them. The operation *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.

Given two aspects asp_1 and asp_2, their corresponding matches on a component $\mathcal{C}_0 = \pi(\gamma(\mathcal{B}))$ are: $m_1 = match_\ell(asp_1.B, asp_1.pc)$ and $m_2 = match_\ell(asp_2.B, asp_2.pc)$ where $asp_1.B, asp_2.B \in \mathcal{B}$. Weaving asp_1 results in $\langle \mathcal{C}_1, m_1 \rangle = weave_\ell(\mathcal{C}_0, asp_1.B, m_1, asp_1.V, asp_1.adv)$. The weaving of asp_2 needs to apply on $\mathcal{C}_1$ and therefore its original match m_2 needs to apply to transitions in $\mathcal{C}_1$ as the local weave changes the transitions. Therefore m_2 is projected onto $\mathcal{C}_1$ using $project(m_2, g_1)$. Which replaces the old transitions with their changes.

$$project(M, m) = \bigcup_{t \in M} (m(t))$$

To enable weaving the k^{th} aspect we need to project over all $k-1$ weaves using $\langle m_1 \ldots, m_{k-1} \rangle$.

$$follow(M, \langle m_1, \ldots, m_{k-1} \rangle) = \begin{cases} project(project(project(M, m_1), \ldots), m_{k-1}) & k-1 > 1 \\ M & \text{otherwise} \end{cases}$$

The *weaveAll* for a sequence of n local aspects on a composite component $\mathcal{C}_0$ is defined as $\langle \mathcal{C}_n, m_n \rangle = weaveAll(\mathcal{C}_0, \langle asp_1, \ldots, asp_n \rangle)$ such that:

$\forall i \in \{1, \ldots, n\}$:

$$\begin{aligned}\langle \mathcal{C}_i, m_i \rangle &= weave_g(\mathcal{C}_{i-1}, follow(M_i, \langle m_1, \ldots, m_{i-1} \rangle), asp_i.V, asp_i.adv) \\ M_i &= match_g(\mathcal{C}_0, asp_i.pc)\end{aligned}$$

Discussion

The reset location in local advices is designed to cause the component to directly move to a location given a guard. In the case of *weaveSerial*, the reset location only incorporates advices of aspects that are woven after it, but not before. The *weaveAll* provides finer tuning on the behavior of a reset location by making reset location not incorporate advices of subsequent aspects. This provides another way to manage interference between various aspects implementing reset location.

For both *weaveSerial* and *weaveAll*, the reset location can cause a component to execute a `before` advice without its corresponding `after`. This scenario typically happens when a reset location is woven on an earlier execution frame than an `after` advice. Given a sequence of aspects $\langle a_1, \ldots, a_n \rangle$ and two aspects a_i and a_j such that $i < j$. Let the frame of a_i's joinpoints be $\langle i_s, i_e \rangle$ and that of a_j's joinpoints be $\langle j_s, j_e \rangle$. If $j_s \prec i_e$ and $i_e \prec j_e$, a reset location in $a_i.adv$ would be woven prior to j_e. Therefore, j_e is not executed along with any after advice woven onto it. It is possible to avoid the scenario by composing aspects differently. Two example compositions involve:

1. Ordering aspects by their execution frames;
2. Splitting an aspect into two aspects such that (a) one contains a reset location only advice and (b) another contains the `before` and `after` advice only. Then, the ones with reset locations are woven first, the rest are woven with *weaveSerial*. This ensures all `before` and `after` apply to all reset locations.

Example 25. *Figure 6.2 shows the different results obtained by composing two aspects a and a' onto an atomic component. The joinpoints are provided by matching $a.pc = \{atLocation(\ell_0), portExecute(p_2)\}$ and $a'.pc = \{atLocation(\ell_1)\}$. The corresponding advices are $a.adv = \langle f_b, f_a, \{\langle \ell_0, true \rangle\} \rangle$ and $a'.adv = \langle f'_b, f'_a, \{\} \rangle$.*

- *Figure 6.2 (a) and Figure 6.2 (b) show the weaving of each aspect individually. When weaving the two aspects into one system, overlap causes interference. Let*

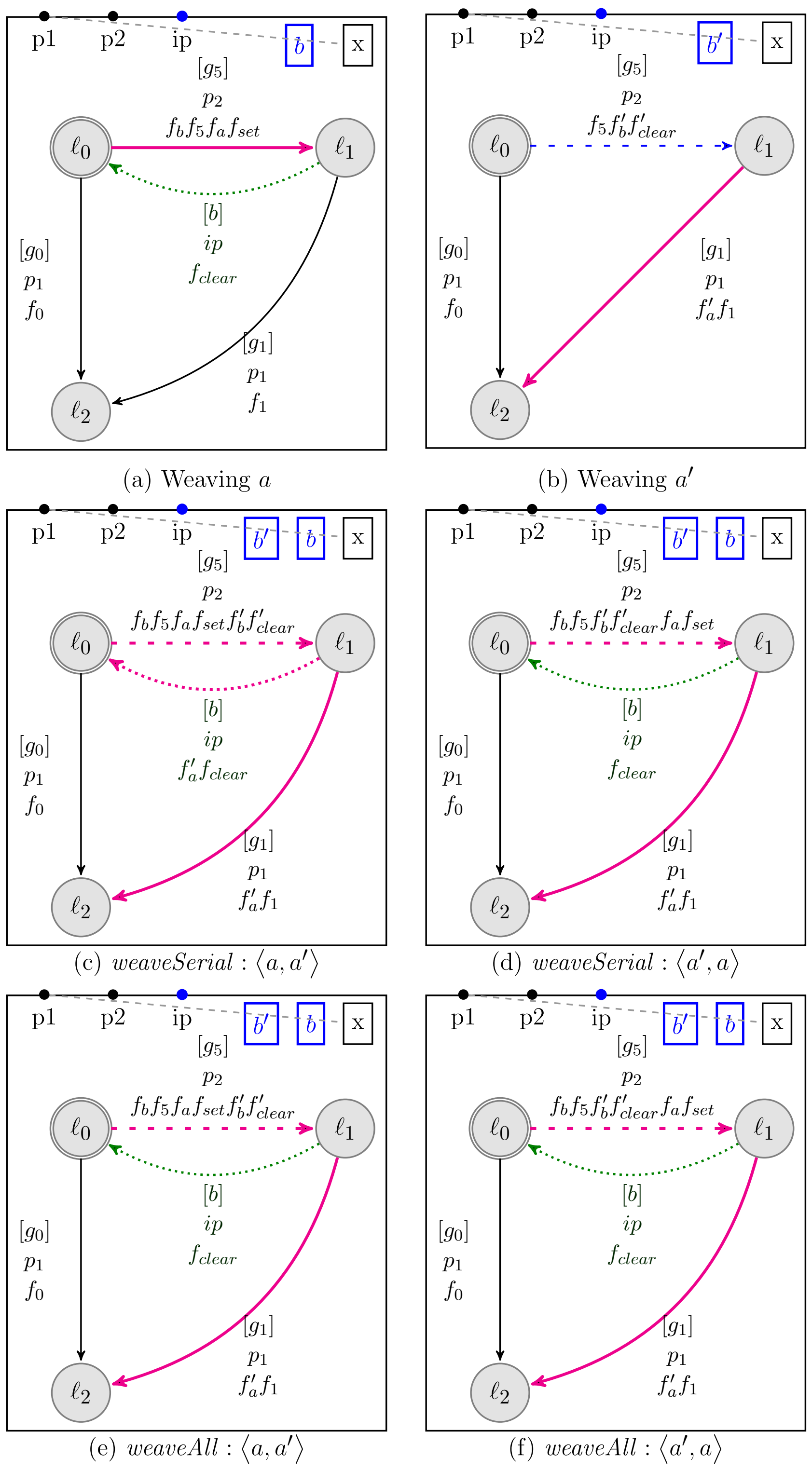

p1
p2
ip
b
b'
x
$[g_5]$
p_2
$f_b f_5 f_a f_{set}$
$f_5 f'_b f'_{clear}$
$f_b f_5 f_a f_{set} f'_b f'_{clear}$
$f_b f_5 f'_b f'_{clear} f_a f_{set}$
ℓ_0
ℓ_1
ℓ_2
$[b]$
ip
f_{clear}
$f'_a f_{clear}$
$[g_0]$
p_1
f_0
$[g_1]$
f_1
$f'_a f_1$
(a) Weaving a
(b) Weaving a'
(c) *weaveSerial* : $\langle a, a' \rangle$
(d) *weaveSerial* : $\langle a', a \rangle$
(e) *weaveAll* : $\langle a, a' \rangle$
(f) *weaveAll* : $\langle a', a \rangle$

Figure 6.2: Weaving Procedures

m denote the match of a, m' denote the match of $a'.pc$ and t_5 the transition guarded by g_5. The transition t_5 is both in the joinpoint of m and previous(m'). Depending on the order of weave the newly created reset location could also be

found in m' as it is outbound from ℓ_1.

- *Figure 6.2 (c) and Figure 6.2 (d) show the weaveSerial$_\ell$ operation.*

  1. *Figure 6.2 (c) presents the serial weave on a followed by a'. The weave of a results in $\langle \mathcal{C}_1, g_1 \rangle$. $\mathcal{C}_1$ is shown in Figure 6.2 (a). 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 $g_1(t_5)$ as it is inbound, therefore it appends $f'_b f'_{clear}$ to its existing function which is now $f_b f_5 f_a f_{set}$ since a was already woven.*

  2. *Figure 6.2 (d) presents the serial weave on a' followed by a. The weave of a' results in $\left\langle \mathcal{C}'_1, g'_1 \right\rangle$. The component is shown in Figure 6.2 (b). Upon weaving a the pointcut will match $g'_1(t_5)$. Its function $f_5 f'_b f'_{clear}$ is then appended with $f_a f_{set}$. Additionally the reset location is then added guarded by $b \wedge true$.*

- *Figure 6.2 (f) show the weaveAll operation and how it differs from the weaveSerial. The pointcut of a' will always match against the original component, matching always the transition guarded by g_5. The weaveAll projects the match. The new match result will be t_5 if a' is woven first or $g_1(t_5)$ if a is woven first. The joinpoint will therefore not contain the reset location in both cases. The order of the aspects still define the order of the advices woven onto overlapping transitions. Figure 6.2 (e) displays the different advices order.*

Chapter 7
AOP-BIP

Contents

7.1 Overview 60
7.2 Language Description 61
7.3 Frontend 66
7.4 Backend 68
7.5 Utility 71

7.1 Overview

The ideas presented in this thesis are implemented in AOP-BIP. AOP-BIPis a proof-of-concept, aspect-oriented extension to BIP. The main flow and various major components are displayed in Figure 7.1 .

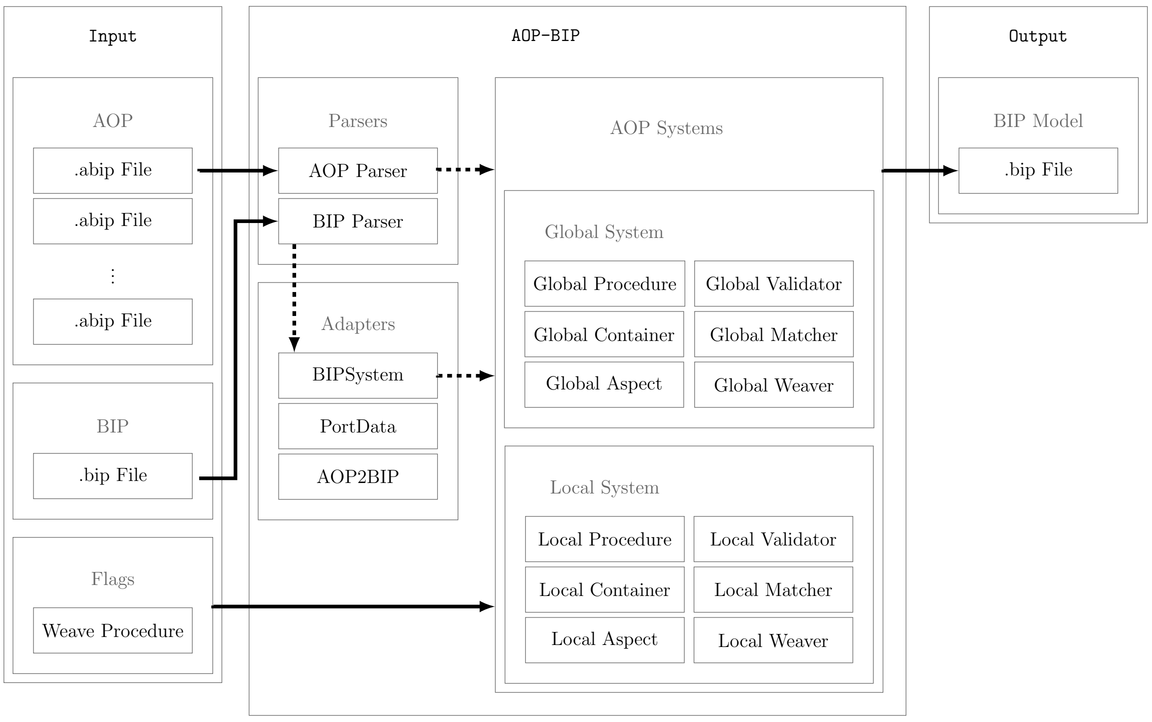


Figure 7.1: The AOP-BIPTool

AOP-BIP's command line front-end takes as input:

- A `.bip` file that represents a BIP system written in the BIP language [11];
- The name of the *Weaving Procedure* to apply when weaving the aspect-oriented descriptions onto the BIP model.
- A collection `.abip` files that represent aspect-oriented descriptions.

AOP-BIPwill then produce 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*. The procedure will then weave the containers onto the BIP model resulting in an output BIP model. A typical weaving procedure will do the following:

- Call the `Validator` to validate the aspect container against the BIP model;
- Call the `Matcher` to match the pointcuts;
- Invoke the `Weaver` to weave in the advices.

The output BIP model is then written to an output file in the BIP language. The command-line client is `ujf.verimag.bip.aop.Main` it can be invoked as:

```
java  -jar  aopbip.jar  bipin  bipout  all|serial  aspect [aspect, ...]
```

The `ujf.verimag.bip.aop.Main` also provides the `weave` static method that provides the same functionality:

```
void weave(String bipin, String bipout, WeaveStrategy strat, Collection<String> input)
```

7.2 Language Description

The BIP Language

The BIP language is described in full in [11]. We only use a subset of the language to illustrate the concepts of this paper. The BIP language is typed. That is, components, ports and data are associated with types. The syntax covering the scope of this thesis is shown in Listing 7.1 . Port types define the type of a port, a type defines the number and corresponding types of variables attached to the port.

```
port type DataPort(int var)
port type VoidPort

atomic type Count (String name)
  data int x
  data int y

  export port DataPort stop(x)
  export port VoidPort tick
  export port VoidPort start

  place L0, L1

  initial to L0 do { x = 0; y = 0; }

  on start  from L0 to L1 provided (x >= 0) do { y++;     }
  on tick   from L1 to L1 provided true     do { x += y;  }
  on stop   from L1 to L0 provided (x > 0)  do {          }
end
```

Listing 7.1: The Counter Component

An atomic type describes an atomic component. It is characterized by:

- A set of parameters, defined upon instantiation;
- A set of variables;
- A set of ports: `export` specifies that the port is an interface, if `export` is omitted the port is internal and executes as if it were connected to a singleton connector;
- A set of control locations;
- An initial location with an initializing function;
- A set of transitions in the form of:

  `on` $<$ `port` $>$ `from` $<$ `location` $>$ `to` $<$ `location` $>$

  `provided` $<$ `guard` $>$ `do` $<$ `function` $>$.

A compound type describes a composite component. A compound type contains a set of components and a list of connectors that define the interactions between components. A connector type defines the port types associated with it, a guard and two functions to execute: `up` and `down`. The `up` defines behavior when the interaction is enabled while `down` defines the behavior when the interaction is executed. Listing

7.2 creates the composite component and the interactions. The interactions ensure that: both counters start and stop in synchrony; `c1` can tick independently while `c2`'s ticking is synchronized with `c1`.

```
connector type Sing(VoidPort t0)
  define [t0]
  on t0 provided true up{} down {}
end
connector type Sync2(VoidPort t1, VoidPort t2)
  define [t1 t2]
  on t1 t2 provided true up {} down {}
end
connector type SwapData(DataPort c1, DataPort c2)
  define [c1 c2]
  on c1 c2 provided true  up{} down{ //Swap
      c1.var = c1.var + c2.var;
      c2.var = c1.var - c2.var;
      c1.var = c1.var - c2.var;
    }
end
compound type Composite
  component Count c1 ("Counter 1")
  component Count c2 ("Counter 2")
  connector Sing  tickOne     (c1.tick)
  connector Sync2 tickAll     (c1.tick, c2.tick)
  connector Sync2 startAll    (c1.start, c2.start)
  connector SwapData stopAll  (c1.stop, c2.stop)
end
component Composite sys
```

Listing 7.2: Composing Two Counters

In addition to 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 will be included in code generation. For example preprocessor directives, type definitions and extra `C` functions are defined in the header. Additionally, functions may be augmented with `C` code by wrapping it in {# #} tags.

The AOP-BIPLanguage

To designate the joinpoint, we begin by illustrating the pointcut expressions. There exists two pointcut expressions based on the type of the aspect: local and global. Listing 7.3 depicts the grammar for local and global aspects. The local pointcuts adopt a similar style to that in the paper. The local advice is a triple of `before`, `after` and an optional reset location pairs.

```
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;
```

Listing 7.3: The Aspects Syntax

The global pointcut syntax definition includes an additional layer `portspec`. The port specification allows us to alias a port identifier. This is merely provided as syntactic sugar to simplify referring to the port variables in the read, write expressions and the advice.

Aspects are grouped into containers so as to encapsulate multiple pointcut expressions and their corresponding advices. The inter-type is defined at the container level so as to encourage encapsulation of shared data. A container is defined by declaring the `Aspect` keyword followed by its identifier. If the container defines local aspects then it must specify the atomic component it targets right after its identifier. The inter-type section lists the extra variables and can optionally specify their initial values. Additionally it is possible to specify extra arbitrary `C` code to merge with the BIP model's header. Listing 7.4 depicts the extra syntax.

```
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
          ;
```

Listing 7.4: The Containers Syntax

Note that syntactically, containers can only either contain global or local aspects, however a file can include a mix of both.

Example 26. *Listing 7.5 lists two containers:* `CyclicTimer` *and* `BalanceTimers`.

1. *The* `CyclicTimer` *container applies two local aspects to* `c1`. *The aspects introduce a new variable* `cycle`. *The first aspect executes after the tick operation and restricts x to the cycle. It also introduces a failsafe, where if it at any point the cycle is set to* 0, *the added reset location forces the system to component to go back to its initial state. In this case this will cause a deadlock stopping the system. The second aspect re-initializes the cycle range whenever* L0 *is entered.*

2. *The* `Catchup` *container applies one global aspect to the system. It introduces a new inter-type that keeps track of the last maximum value. The aspect is applied to any interaction both containing the two stop ports for the timers and data is sent to their* `var` *variable. The first aspect happens after the timers have stopped and swapped their values. It forces the timers to synchronize their value to their current maximum or the history's maximum.*

```
{#
    #define MAX(X, Y) (((X) > (Y)) ? (X) : (Y))
#}
Aspect CyclicTimer (c1) {
  data int cycle = 1
  portExecute(tick) do
    {}                   //Before
    {x = x % cycle;}     //After
    {(L0, cycle == 0)}   //Reset

  atLocation(L0) do {cycle = y * 2;} {}
}
Aspect BalanceTimers {
  data int lastmin = -1;
  ports(p1:c1.stop, p2:c2.stop)
    writePortVars(p1.var, p2.var)
  do
  {}
  {
    p1.var  = MAX(lastmin, MAX(p1.var, p2.var));
    p2.var  = p1.var;
    lastMin = p1.var;
  }
}
```

Listing 7.5: Example Aspects

We select an atomic component or an interaction by its *identifier*, and not its type. 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 identifiers specified in the AOP model generated from an `.abip` are expected to reference identifiers in the BIP model generated from the `.bip` file. The `validators` are responsible for verifying that the identifiers match.

7.3 Frontend

Loading The Models

The `BIPSystem` class plays the role of an adapter of the BIP model. It provides the interface to `load` the `.bip`, construct the BIP model and `save` it to a file. This is accomplished by interaction with the existing BIP framework tools. The `BIPSystem` is also used to query the BIP model: find the corresponding atomic type for an identifier and return all components. It can also be used to perform model-level edits such as cleaning up the types no longer used and cloning the types.

The `AspectLoader` is responsible for parsing the `.abip` files and generating the corresponding containers. Parsing is done using Antlr4 [18]. It typically reads an `.abip` file and return an `AspectFile` which contains a header and a list of containers.

Aspects

At the very core of the entire tool lie the `Aspect` interface. it determines the operations permissible on all types of aspects. Two types of aspects are defined: `GlobalAspect` and `LocalAspect`. Figure 7.2 depicts the `Aspect` interface. The extra operations relative to retrieving additional information are shown for `LocalAspect` and `GlobalAspect`. A local aspect applies to a component, therefore it must return its identifier. A global aspect defines a set of aliased ports in its *ports* pointcut. An `AspectContainer` is parametrized by either `LocalAspect` or `GlobalAspect`, it defines an order over the aspects and has a name. The base classes for containers are `LocalAspectContainer` and `GlobalAspectContainer` they implement `AspectContainer` < `LocalAspect` > and `AspectContainer` < `GlobalAspect` > respectively.

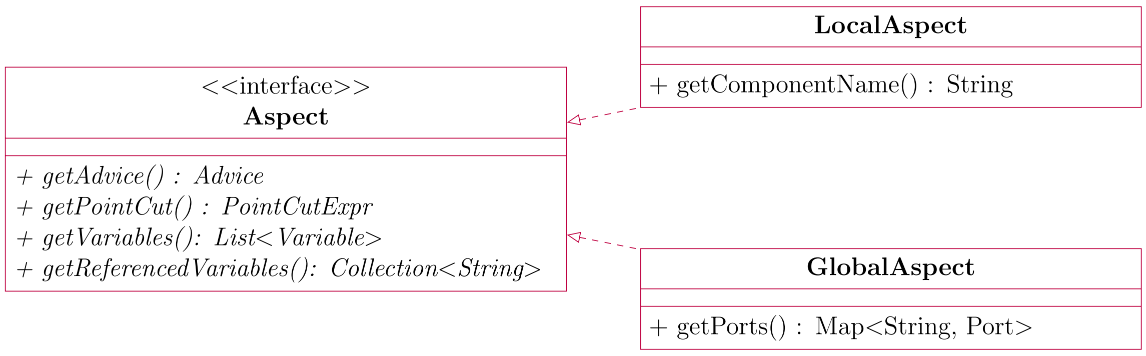

LocalAspect
+ getComponentName() : String
<<interface>>
Aspect
+ getAdvice() : Advice
+ getPointCut() : PointCutExpr
+ getVariables(): List<Variable>
+ getReferencedVariables(): Collection<String>
GlobalAspect
+ getPorts() : Map<String, Port>

Figure 7.2: The Aspect Interface

Interacting with the AOP System

An aspect is typically disassociated from any information about the BIP model. All higher constructs are parameterized in terms of the aspect type. The `AOPSystem` defines the interface that manages the interaction between the aspect container and a BIP system. Classes implementing the `AOPSystem` interface are `GlobalAOPSystem` and `LocalAOPSystem` The `AOPSystem` interface defines two operations:

1. `valid`: validates an aspect container by cross-referencing the identifiers;
2. `match`: matches all aspects in the container and return their corresponding `Joinpoint` objects.

The `AOPSystem` interface does not define an interface for weaving, weaving is handled at an upper level by `WeaveProcedure` which makes use of the `AOPSystem`. The `AOPSystem` keeps track of partial weaving information and history but does not invoke directly the weaver. Its responsibility is to keep track of both the original BIP model and the changes to it during a weave. Two weave procedures are presented: `LocalWeaveProcedure` and `GlobalWeaveProcedure`. They implement `weaveSerial` and `weaveAll` as described in Section 6.3. Listing 7.6 displays the weaving of one local container onto a BIP model.

```
BIPSystem sys     = new BIPSystem("/path/to/bip");
AspectFile file   = AspectLoader.load("/path/to/abip");
//Track Created Objects
List<Object> artifacts              = new LinkedList<Object>();
//Assuming Local Aspects
LocalAspectContainer container      = (LocalAspectContainer)
                                      file.getContainers().get(0);
AOPSystem<LocalAspect> aopsys       = new LocalAOPSystem(sys);
WeaveProcedure<LocalAspect> proc    = new LocalWeaveProcedure();
//Use Weave Serial
proc.weaveSerial(sys, container, artifacts);
//Save
sys.save("/path/to/output/file");
```

Listing 7.6: A Simple Weaving

7.4 Backend

Conversion

The `AOP2BIP` class handles conversion from the AOP model to the BIP model. It is used by the weavers to perform the necessary edits onto the BIP model. Operations include:

- Creating extra BIP variables (for inter-type);

- Advice code conversion from AOP to BIP;
- Linking variables in AOP to their corresponding BIP variables;
- Merging code headers;
- Extending connectors by additional ports;
- Creating the component for the global inter-type.

Matching

The `Matcher` has two responsibilities: handling the matching of joinpoints and providing the search in the corresponding BIP model. The `LocalMatcher` for example provides `getOriginSet` which returns the origin set for a list of transitions. It also performs checking the guard expressions or the function code for variables.

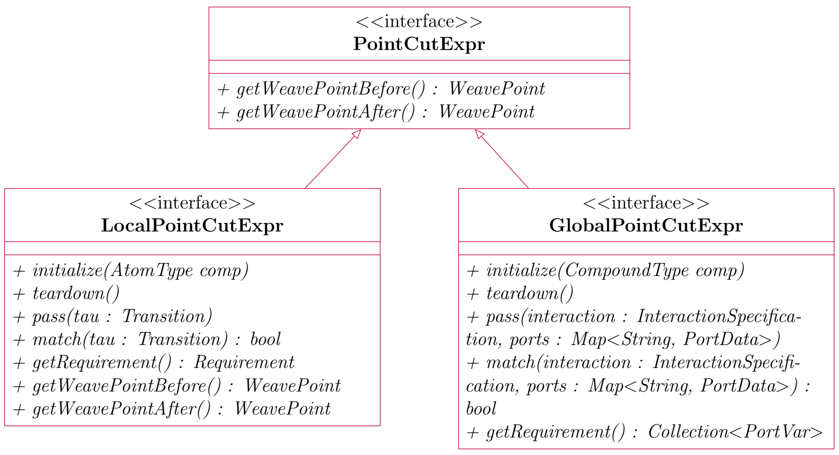

<<interface>>
PointCutExpr
+ getWeavePointBefore() : WeavePoint
+ getWeavePointAfter() : WeavePoint
<<interface>>
LocalPointCutExpr
+ initialize(AtomType comp)
+ teardown()
+ pass(tau : Transition)
+ match(tau : Transition) : bool
+ getRequirement() : Requirement
+ getWeavePointBefore() : WeavePoint
+ getWeavePointAfter() : WeavePoint
<<interface>>
GlobalPointCutExpr
+ initialize(CompoundType comp)
+ teardown()
+ pass(interaction : InteractionSpecification, ports : Map<String, PortData>)
+ match(interaction : InteractionSpecification, ports : Map<String, PortData>) : bool
+ getRequirement() : Collection<PortVar>

Figure 7.3: The Pointcut Interface

The `PointCutExpr` interface shown in Figure 7.3 , represents the pointcut expression. A class implementing it must implement the matching logic. The matcher will first call `initialize` with an atomic component in the case of local or the composite component in the case of global. The matcher goes through all transitions in the component twice. The first pass executes `pass` and is used to gather information prior to matching. The second pass executes `match` and is used to filter the interactions, if match returns true the interaction is kept in the match. Typically the first pass finds the exact match and the second pass matches its *siblings*. After matching, it invokes `teardown` for cleanup. Listing 7.7 displays the matching performed by the `ReadVarGuard` pointcut.

```
HashSet<String> states;
public void initialize(AtomType comp) {
  states = new HashSet<String>();
}
public void teardown() {
  states.clear();
}
public void pass(Transition tau) {
  Expression e = (Expression) tau.getGuard();
  if(LocalMatcher.findVar(e, name))
    states.add(tau.getOrigin().get(0).getName());
}
public boolean match(Transition tau) {
  return states.contains(tau.getOrigin().get(0).getName());
}
```

Listing 7.7: ReadVarGuard Matching

Weaving

The `WeaveType` provides necessary history when weaving. It has three responsibilities:

1. It keeps track of the AOP objects: the ports relevant to AOP, the additional boolean variable, the inter-type component (for global weaving).

2. It keeps the original type and the type being woven onto separate during the weave procedure.

3. It provides necessary operations to map the matches from the original type to the edited type.

The `Weaver` provides operations to edit the BIP model at a high level. It implements the weaving rules. The `LocalWeaver` for example, provides operations like: `weaveAdd` and `weaveReset`[1]. It implements weaving of *one* aspect. The `WeaveProcedure` implements the weaving of *multiple* aspects enclosed in an *aspect container*.

1. As described in Sections 4.6.4 and 4.6.5

```java
public void weaveSerial(AOPSystem<LocalAspect> system,
  AspectContainer<LocalAspect> aspects, List<EObject> artifacts)
  throws AopException {
  //Validate
  List<String> messages = new LinkedList<String>();
  if(!system.valid(aspects, messages)) return;
  //Container used to match a single aspect
  AspectContainer<LocalAspect> con  = new LocalAspectContainer();
  for(LocalAspect aspect : aspects){
    //Match the single aspect against the new system
    con.clear();
    con.add(aspect);
    Map<LocalAspect, JoinPoint> matches = system.match(con);
    LocalJoinPoint jp    = (LocalJoinPoint) matches.get(aspect);
    //Matched
    if(!jp.getMatches().isEmpty()) {
      LocalWeaveType weave = ((LocalAOPSystem) system)
                          .getWeaveType(aspect);
      //Create a new AOP variable
      weave.renewVariable();
      //Weave inter-type if necessary (existing variables are untouched)
      Aop2Bip.createVariables(weave.getTargetType(), aspect, artifacts);
      //Weave the Aspect
      LocalWeaver.weaveAdvice(aspect, jp, weave, artifacts);
      //Commit changes
      system.getBIP()
              .getComponent(aspect.getComponentName())
              .setType(weave.getTargetType());
    }
    //Cleanup History (new weaves operate on new type)
    ((LocalAOPSystem) system).cleanup();
  }
}
```

Listing 7.8: Local Serial Weave

7.5 Utility

Three classes are provided for utility functions:

1. The `Naming` class is responsible for handling the type names and identifier names of the created types. It determines the naming convention used for transforming the BIP model. For example `StringcopyType(AtomTypetype)`, returns the name of the new copied type of an atomic component.

2. The `Util` class provides some extra functionality for labeling transitions in a BIP model.

3. The `TestParser` class provides an executable main function that takes in an `.abip` file and displays its concrete syntax tree using *Antlr4*'s inspection tool.

Chapter 8
CASE STUDY

Contents

8.1 Overview 73
8.2 Logging Concern 74
8.3 Security Concerns 75
8.4 Performance Concerns 76
8.5 Fault Tolerance Concerns 77
8.6 Using Inter-type Structures 77

8.1 Overview

A sample network protocol is used to illustrate crosscutting concerns in BIP. Figure 8.1 presents the `Network` composite component which is composed of a `Server`, `Client` and a `Channel`.
The network protocol is an augmented version of the one presented in [19]. The protocol is as follows:

1. The `Server` waits for the *clear-to-send* signal on its `cts` port. This indicates that a channel is available.

2. The `Server` generates a packet and sends it to the `Channel`.

3. The `Channel` forwards the packet to client.

4. The `Client` acknowledges the packet by sending an acknowledgment.

5. The `Channel` forwards the acknowledgment back to the `Server`.

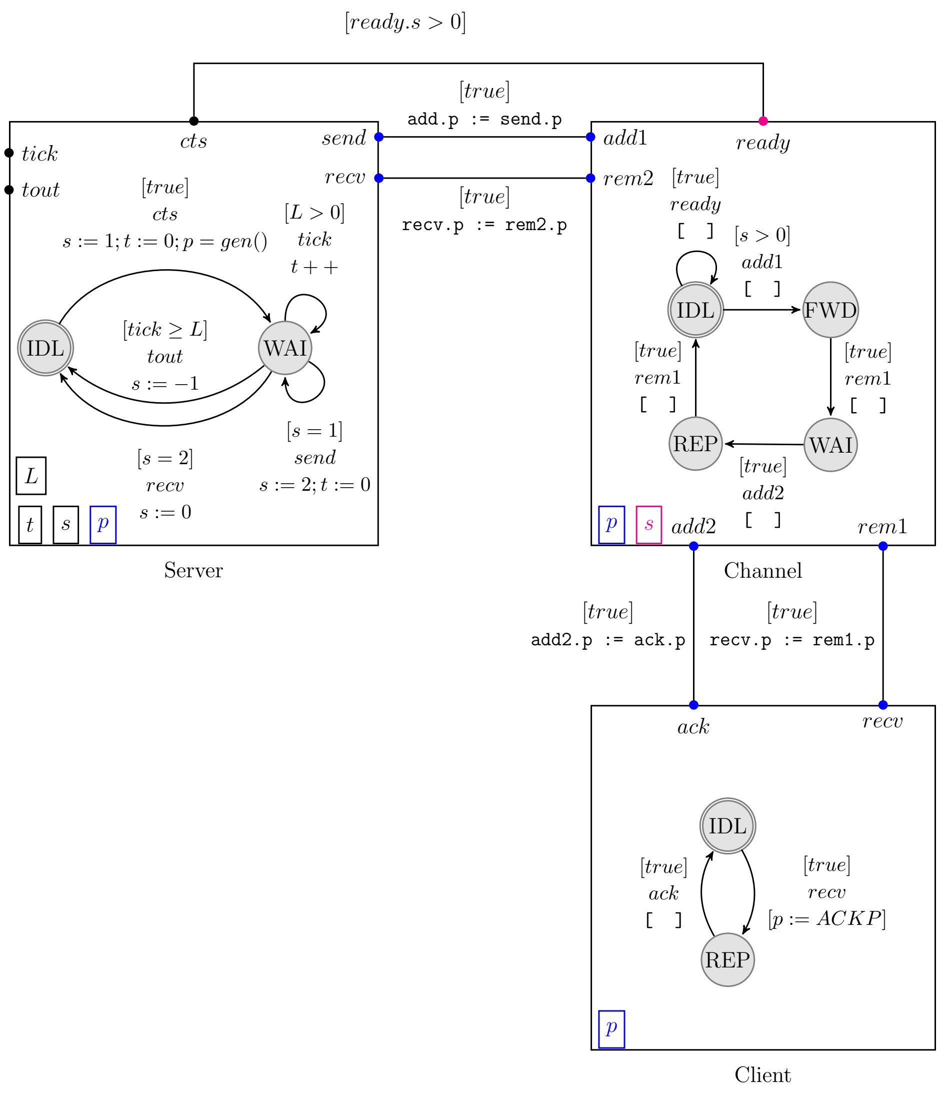


Figure 8.1: The `Network` Component

8.2 Logging Concern

Running the initial version of the BIP model produces no output. We begin by adding logging. The logging can be achieved either by logging the interactions or the individual components. We will adopt the latter, as we are interested to see the inner changes. To do so, we match all port executions in `Server`, `Client` and `Channel`. The standard output is used for logging.

```
{# const char * val(NPacket p) { return p.c_str(); } #}
Aspect LogServer(server) {
  portExecute(cts)
  do {}{ printf("[%s] Clear to Send\n", id, val(p)); }

  portExecute(send)
  do { printf("[%s] -> %s (Time: %d)\n", id, val(p), t); } {}

  portExecute(recv)
  do { printf("[%s] <- %s (Time: %d)\n\n\n", id, val(p), t);} {}

  portExecute(tout)
  do {}{ printf("[%s] Timeout\n", id); }
}
Aspect LogClient(client) {
  portExecute(recv)
  do {} {printf("[%s] <- %s\n", id, val(p)); }

  portExecute(ack)
  do {} {printf("[%s] -> ACK\n", id); }
}
Aspect LogChannel(channel) {
  portExecute(add1)
  do {printf("[Channel] <- %s\n", val(p)); } {}

  portExecute(rem1)
  do {printf("[Channel] -> %s\n", val(p)); } {}

  portExecute(add2)
  do {printf("[Channel] <- %s\n", val(p)); } {}

  portExecute(rem2)
  do {printf("[Channel] -> %s\n", val(p)); } {}
}
```

```
[Server] Clear to Send
[Server] -> 549 (Time: 0)
[Channel] <- 549
[Channel] -> 549
[Client] <- 549
[Client] -> ACK
[Channel] <- ACK
[Channel] -> ACK
[Server] <- ACK (Time: 0)


[Server] Clear to Send
[Server] -> 78 (Time: 0)
[Channel] <- 78
[Channel] -> 78
[Client] <- 78
[Client] -> ACK
[Channel] <- ACK
[Channel] -> ACK
[Server] <- ACK (Time: 5)
```

Listing 8.1: Logging Aspects

8.3 Security Concerns

The `channel` presented is a simple channel. It only forwards packets it receives. One common non-functional requirement which domain is crosscutting is *security*. Listing 8.2 introduces authentication to the channel. We seek to authenticate the connection between the `server` and `channel` only. To do so, we add an extra transition from `FWD` to `IDL` in case of failure. Two aspect containers introduce *hash* authentication. The third container simulates a man-in-the-middle attack by intercepting and faking the packet. We use a simple hash function for illustration, the hash of a string is its last character. The `server` computes and appends the hash of its packet when it is ready to send. The `channel` checks the hash and updates its new `clear` variable then removes the hash altogether.

```
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)}
}
//Example Man-in-the-middle
Aspect Carol {
  ports(a:server.send b:channel.add1)
  readPortVars(a.r)
  do {} {b.r = pfake(a.r);}
}
```

```
[Server]  -> 886|6 (Time: 0)
[Channel]  <- 386|6
[Channel]  -> 386
[Client]  <- ACK
[Client]  -> ACK
[Channel]  <- ACK
[Channel]  -> ACK
[Server]  <- ACK (Time: 3)

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

Listing 8.2: Authentication Aspects

8.4 Performance Concerns

Another common non-functional requirement which domain is crosscutting is *performance*. Listing 8.3 introduces a congestion avoidance [20] mechanism to the `server`. This will cause it to avoid flooding the channel in the occurrence of timeouts. A very simple algorithm is used to maintain clarity. The algorithm implemented depends on computing *round-trip-time* (RTT); the sum of the time spent by the server to send and receive the packet. RTT is computed in *ticks*, and only the last RTT of a successful receive is kept. The `server` will wait $twait = RTT - 6$ ticks for any `RTT` higher than `6` and in case of timeout, will double its `RTT` value.

```
Aspect Throttle (server) {

  data int rtt = 1
  data int time2send = 0

  portExecute(send)
  do {time2send = t;} { }
  portExecute(recv)
  do {rtt = time2send + t + 1 ;} {}
  portExecute(tout)
  do {rtt *= 2;} {}
  portExecute(cts)
  do {}{} {(IDL, (rtt > 6))}

  atLocation(IDL)
  do  {rtt--;}
      { printf("[%s] RTT: %d\n", id, rtt);}
}
```

```
[Server]  RTT: 5
[Server]  Clear to Send
[Server]  -> 763|3 (Time: 0)
[Channel]  <- 736|3
[Server]  Timeout
[Server]  RTT: 9
[Server]  Clear to Send
[Server]  RTT: 8
[Server]  Clear to Send
[Server]  RTT: 7
[Server]  Clear to Send
[Server]  RTT: 6
[Server]  Clear to Send
[Server]  -> 281|1 (Time: 0)
[Channel]  <- 925|1
```

Listing 8.3: Congestion Avoidance Aspects

8.5 Fault Tolerance Concerns

Listing 8.4 depicts the `Failsafe` concern. If at any point the `channel` receives four extra packets than it had delivered, it shuts down. We simply keep track of the number of sent packets and the number of received packets. We do it by monitoring the channel externally. The only port accessible that could alter the channel's `state` is `channel.ready`. When checking for a ready, we verify the failure, and if detected, set the channel's state to closed. The server then times out and never receives the `cts` signal. The system deadlocks and terminates gracefully.

```
{#
  #define THR 3
  int checkFail(int status, int s, int r) {
      return (s - r) >= THR ? -1 : status;
  }
#}
Aspect Failsafe {
  data int sent
  data int received

  ports(a:server.send b:channel.add1)
  do {} {sent++;}

  ports(a:server.recv b:channel.rem2)
  do {} {received++;}
  //Disable the channel
  ports(a:channel.ready)
  do {} {a.r = checkFail(a.r, sent, received);}
}
```

```
[Server] Clear to Send
[Server] -> 980|0 (Time: 0)
[Channel] <- 862|0
[Server] Timeout
[Server] Clear to Send
[Server] -> 932|2 (Time: 0)
[Channel] <- 12|2
[Server] Timeout
[Server] Clear to Send
[Server] -> 927|7 (Time: 0)
[Channel] <- 856|7
[Server] Timeout
[Server] Clear to Send
[Server] Timeout
scheduler deadlock!
```

Listing 8.4: Failsafe Aspects

8.6 Using Inter-type Structures

Listing 8.5 depicts a set of aspects that monitors a channel, tracking its history. A stack is used to keep a history of packets sent to the channel. The channel will then buffer the packets until delivery and will do some analysis. In this case our analysis will count odd and even numbered packets in the history.

```
Aspect History {
  data History hist

  ports(a:server.send b:channel.add1)
  do {push(&hist, a.r);} {}

  ports(a:server.recv b:channel.rem2)
  do {analyze(&hist); }  {}
}
```

```
[Client] <- ACK
[Client] -> ACK
[Channel] <- ACK
[History] 16 odd, 8 even
[Channel] -> ACK
[Server] <- ACK (Time: 1)
```

Listing 8.5: History Aspects

Chapter 9
RELATED WORK

Contents

9.1 Aspect-Oriented and Component-Based Design Integration 78
9.2 Integration of Aspect-Oriented concepts in Automata 81

9.1 Aspect-Oriented and Component-Based Design Integration

Pessemier [10] presents a framework to deal with crosscutting concerns in component-based systems using a component-based approach. The implementation of a concern is found in an *Aspect Component*. Aspect components are regular components augmented with an extra interfaces. They contain the advices necessary to implement a crosscutting concern. Interaction with the advice happens through additional interfaces known as *advice interfaces*. Moreover, Aspects components expose regular interfaces. Thus, they can be seen as regular components. This model allows component-based approach to be used when maintaining and developing components tackling crosscutting concerns. A joinpoint is of two types: incoming and outgoing calls on a corresponding interface. A set of joinpoints consists of a combination of selected interfaces in different components. To select interfaces, pointcut expressions consist of two main sections: the first section determines whether its an incoming (CLIENT), an outgoing (SERVER) or both incoming and outgoing call; the second section is a set of three regular expressions that capture the interface signature. The interception model is based on composition filters [21] but extended from objects to components. It considers a component a black box and therefore only intercepts incoming and outgoing calls to it. An incoming filter is placed on incoming calls to a component. The filter is capable of either forwarding the calls destined to the component or blocking them. An outgoing filter is placed on outgoing calls to a component. The filter is capable of either forwarding the calls originating from the component or blocking them.

After selecting the interfaces, each interface is bound to the corresponding *aspect component* using an *aspect binding*, therefore executing the appropriate advice through the component's interface. The selected interfaces and their *aspect bindings* to a specific *aspect component* are logically enclosed in an *aspect domain.* The purpose of the *aspect domain* is to keep track of the affected components. Therefore, for each crosscutting concern coincides one *aspect domain.* Figure 9.1 depicts the various relationships between *aspect components*, *aspect bindings* and *aspect domains.*

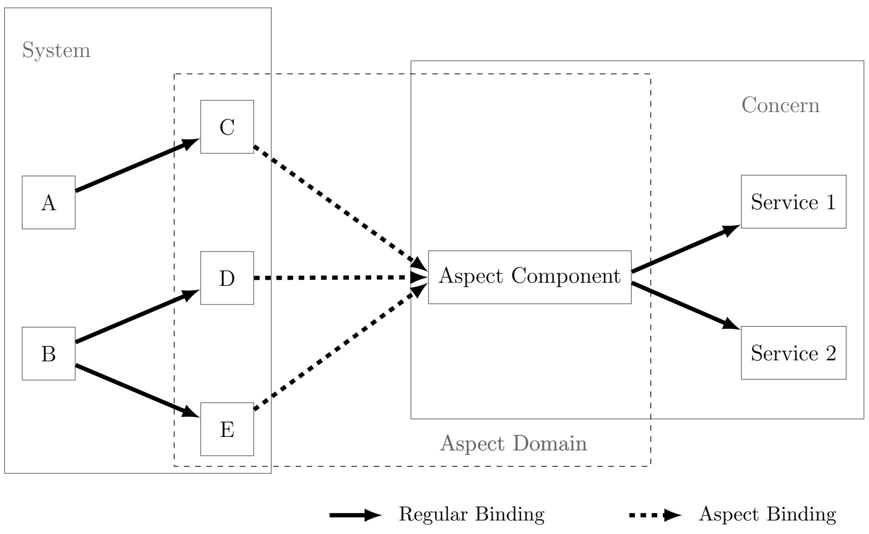


Figure 9.1: Aspect Components, Aspect Bindings and Aspect Domains

The model is mapped onto Fractal, a modular and extensible component model [22]. Additionally, weaving is determined at runtime. This allows the system to reconfigure by enabling or disabling specific aspects at runtime. This approach explicitly models dependencies between aspects and components, and allows for their composition at an architectural level.

This approach contains similar notions found in ours. The interception of the interactions between individual components views components as black boxes. In our approach, global aspects target the interaction between components regarding them as black boxes. The exposed information is the interface and its data transfer. A global pointcut targets the *ports* and whether or not their variables have been written or read. The *ports* pointcut selects the ports, which is similar to the three regular expressions selecting the interfaces. The combination of *readPortsVars* and *writePortsVars* can express CLIENT, SERVER and both CLIENT and SERVER matching. The expression *readPortsVars* detects incoming data transfer while *writePortsVars* detects outgoing data transfer.

However, the approach is symmetric. Aspects are represented using component-based concepts. The joinpoint, pointcut, weaving information is not lost after the weave, but represented as part of the system. This representation confers several advantages. First, it allows the aspects to be manipulated and reconfigured at runtime. Second, it clearly defines the relationships between the aspects and other aspects, and aspects and the components they modify. Third, it reduces the efforts needed to maintain the system. Since aspects and non-aspect components have the same representation, the system is maintained as one. Asymmetrical approaches require knowledge both in the component-based approaches and the aspect representation. This approach however does not take into account the semantics of the interaction. An advice is an arbitrary code execution, a pointcut is an arbitrary function call. 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. In approaches targeting arbitrary interface signatures, the implementation itself must explicitly address the issues of synchronization amongst the different components and data transfer.

The approach stated is an elaboration on other works such as [8],[9], and [23] which have been done to integrate AOP into CBS systems as well. Duclos' approach [8] defines two languages. The *Aspect Definition Language* describes advices at an architectural level and the *Aspect User Language* which defines how the aspects are applied into the CBS system. Lieberherr's approach [9] defines aspects as part of the modules they apply to, and compares the expressiveness of the approach with both AspectJ and HyperJ [24]. SAFRAN [23] uses AOP in the Fractal component model to define adaptation policies. SAFRAN is asymmetric and mostly targets adaptation and reconfiguration as a crosscutting concern. It is therefore less general purpose.

To the best of our knowledge, we have not seen major work on formalizing aspects in component-based frameworks. Furthermore, BIP confers numerous advantages over other component-based frameworks. First, it has a well-defined operational semantics. This allows aspects to be formalized, and thus ensures correct-by-construction systems. Second, it has a strong expressive synchronization primitive which does not exist in other component-based frameworks. It has been shown in [25] that BIP synchronization is more expressive than both CCS [26] and CSP [27]. This allows more concerns to be formalized. Third, it makes a clear separation between behaviors and coordination, providing a clear distinction between two types of aspects (local and global). This distinction is aligned with the component-based paradigm.

9.2 Integration of Aspect-Oriented concepts in Automata

Larissa [28] is a language for handling crosscutting concerns in reactive systems. Systems are modeled as the parallel 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.

The underlying model does not have a clear distinction between communications and components. Components include direct primitive related to the communication model. Moreover, the communication model is based on simple input/output matching. On the other hand, the proposed approach considers only one type of aspect and does not distinguish between aspect related components and aspects related to communications. Consequently, this breaks component-based approach. Finally, advices are not expressive and only consider reset/restore the state of the system.

Chapter 10
CONCLUSION AND FUTURE WORK

Contents

10.1 Conclusion 82
10.2 Future Work 83

10.1 Conclusion

Crosscutting concerns are often found both spread across the entire system and tangled at one point of execution. Handling these concerns in component-based systems helps reduce error and improves maintenance efforts by promoting encapsulation and separation of concerns.

This thesis proposes a framework for handling crosscutting concerns in the component-based framework BIP. There are two stages when dealing with CBSs: (1) the development of the components themselves; and (2) the composition of these components.

Our method enriches the stages of CBSs by defining local and global aspects to refine components and their compositions, respectively. Local pointcuts express execution points found in local components such as entering locations, executing ports. They are associated with advices to inject extra functionalities (e.g., computation or change the location of the component). Global aspects are applied to the interaction of components through their interfaces only. That is, without having information about the internal representation of components. Global pointcuts express various ways to match interactions, by considering participating ports and their respective data transfer operations. Similar to a local pointcut, a global pointcut is associated with a global advice. A global advise injects extra functionalities to the interaction model (e.g., data transfer, storing global information, etc.). Local and global aspects have been mapped to BIP semantics. Pointcut matching and advice weaving are implemented by applying model-to-model transformation on BIP models. Consequently, this work makes a new contribution towards correct-by-construction system design. Furthermore, to remedy interference and to increase expressiveness, we present two ways to compose multiple aspects: `weaveSerial` and `weaveAll`.

We implement the proposed method in AOP-BIPtool-chain. It presents a language to describe both local and global aspects and provide 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.

10.2 Future Work

Future work comprises four directions:

- The first direction is to define the semantics for the *aspect* and the transformation itself. Currently we only define the transformation from the BIP structure to another BIP structure. Doing so enables us to restrict the aspects based on the properties they preserve. Work to define aspect categories and the properties they preserve has been done in [29].

- The second direction is to extend both the reach of pointcuts to capture more joinpoints and advices. Currently we can only capture existing transitions locally, and interactions globally. The matching can be extended to add new interactions to track global state. It could also be designed to handle a composition of local joinpoints spread across multiple atomic components in the system. We also plan to extend advices to allow different changes mostly improving recovery. Work on that could be inspired from the notion of masking, pure components and component-based recovery defined in [19].

- The third direction deals with improving our weaving techniques to implement CBS concepts. One approach is to implement advices in separate components. Thus advices may interact and composed in a CBS fashion. This is similar to the notion of aspect component, binding and domain defined in [10] and inspired by [9].

- The fourth direction is to implement model-to-model transformation using a Domain Specific Language (DSL) targeting the BIP model inspired by ATL [30], and compare the expressiveness with our approach.

References

[1] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, “Rigorous component-based system design using the bip framework,” *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *ECOOP*, 1997, pp. 220–242.

[3] X. Coporation. (2003) The aspectj (tm) programming guide. https://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

[4] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972. http://doi.acm.org/10.1145/361598.361623

[5] K. Czarnecki, U. W. Eisenecker, and P. Steyaert, “Beyond objects: Generative programming,” in *CES97a [1 46]. THE 23RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING.* Citeseer, 1997, pp. 5–14.

[6] M. Noureddine, M. Jaber, S. Bliudze, and F. A. Zaraket, “Reduction and abstraction techniques for BIP,” in *Formal Aspects of Component Software - 11th International Symposium, FACS 2014, Bertinoro, Italy, September 10-12, 2014, Revised Selected Papers*, ser. Lecture Notes in Computer Science, I. Lanese and E. Madelaine, Eds., vol. 8997. Springer, 2014, pp. 288–305. http://dx.doi.org/10.1007/978-3-319-15317-9_18

[7] Y. Falcone, M. Jaber, T. Nguyen, M. Bozga, and S. Bensalem, “Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation,” *Software and System Modeling*, vol. 14, no. 1, pp. 173–199, 2015. http://dx.doi.org/10.1007/s10270-013-0323-y

[8] F. Duclos, J. Estublier, and P. Morat, “Describing and using non functional aspects in component based applications,” in *AOSD*, 2002, pp. 65–75. http://doi.acm.org/10.1145/508386.508394

[9] K. J. Lieberherr, D. H. Lorenz, and J. Ovlinger, "Aspectual collaborations: Combining modules and aspects," *Comput. J.*, vol. 46, no. 5, pp. 542–565, 2003. http://dx.doi.org/10.1093/comjnl/46.5.542

[10] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye, "A component-based and aspect-oriented model for software evolution," *IJCAT*, vol. 31, no. 1/2, pp. 94–105, 2008. http://dx.doi.org/10.1504/IJCAT.2008.017722

[11] Verimag. (2015) Bip tools. http://www-verimag.imag.fr/BIP-Tools,93.html

[12] T. E. Foundation. (2015) Eclipse modeling project. Accessed on Augest 13, 2015. https://eclipse.org/modeling/emf/

[13] S. Bensalem, A. Griesmayer, A. Legay, T. Nguyen, J. Sifakis, and R. Yan, "D-finder 2: Towards efficient correctness of incremental design," in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617. Springer, 2011, pp. 453–458. http://dx.doi.org/10.1007/978-3-642-20398-5_32

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, ser. Lecture Notes in Computer Science, J. L. Knudsen, Ed., vol. 2072. Springer, 2001, pp. 327–353. http://dx.doi.org/10.1007/3-540-45337-7_18

[15] M. Bozga, M. Jaber, and J. Sifakis, "Source-to-source architecture transformation for performance optimization in BIP," *IEEE Trans. Industrial Informatics*, vol. 6, no. 4, pp. 708–718, 2010. http://dx.doi.org/10.1109/TII.2010.2069102

[16] K. Kiviluoma, J. Koskinen, and T. Mikkonen, "Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, L. L. Pollock and M. Pezzè, Eds. ACM, 2006, pp. 181–190. http://doi.acm.org/10.1145/1146238.1146259

[17] K. J. Lieberherr and I. M. Holland, “Formulations and benefits of the law of demeter,” *SIGPLAN Notices*, vol. 24, no. 3, pp. 67–78, 1989. http://doi.acm.org/10.1145/66083.66089

[18] T. Parr. (2014) Antlr homepage. http://www.antlr.org/

[19] B. Bonakdarpour, M. Bozga, and G. Gößler, “A theory of fault recovery for component-based models,” in *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings*, ser. Lecture Notes in Computer Science, A. W. Richa and C. Scheideler, Eds., vol. 7596. Springer, 2012, pp. 314–328. http://dx.doi.org/10.1007/978-3-642-33536-5_31

[20] V. Jacobson, “Congestion avoidance and control,” in *SIGCOMM*, 1988, pp. 314–329. http://doi.acm.org/10.1145/52324.52356

[21] M. Aksit, L. Bergmans, and S. Vural, “An object-oriented language-database integration model: The composition-filters approach,” in *ECOOP ’92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands, June 29 - July 3, 1992, Proceedings*, ser. Lecture Notes in Computer Science, O. L. Madsen, Ed., vol. 615. Springer, 1992, pp. 372–395. http://dx.doi.org/10.1007/BFb0053047

[22] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani, “An open component model and its support in java,” in *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, ser. Lecture Notes in Computer Science, I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, Eds., vol. 3054. Springer, 2004, pp. 7–22. http://dx.doi.org/10.1007/978-3-540-24774-6_3

[23] P. David and T. Ledoux, “An aspect-oriented approach for developing self-adaptive fractal components,” in *Software Composition, 5th International Symposium, SC 2006, Vienna, Austria, March 25-26, 2006, Revised Papers*, ser. Lecture Notes in Computer Science, W. Löwe and M. Südholt, Eds., vol. 4089. Springer, 2006, pp. 82–97. http://dx.doi.org/10.1007/11821946_6

[24] P. Tarr and H. Ossher, “Hyper/j: Multi-dimensional separation of concerns for java,” in *Proceedings of the 23rd International Conference on Software*

Engineering, ser. ICSE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 729–730. http://dl.acm.org/citation.cfm?id=381473.381615

[25] S. Bliudze and J. Sifakis, "A notion of glue expressiveness for component-based systems," in *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, ser. Lecture Notes in Computer Science, F. van Breugel and M. Chechik, Eds., vol. 5201. Springer, 2008, pp. 508–522. http://dx.doi.org/10.1007/978-3-540-85361-9_39

[26] R. Milner, *Communication and concurrency*, ser. PHI Series in computer science. Prentice Hall, 1989.

[27] C. A. R. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[28] K. Altisen, F. Maraninchi, and D. Stauch, "Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework," *Sci. Comput. Program.*, vol. 63, no. 3, pp. 297–320, 2006. http://dx.doi.org/10.1016/j.scico.2005.12.001

[29] S. D. Djoko, R. Douence, and P. Fradet, "Aspects preserving properties," *Sci. Comput. Program.*, vol. 77, no. 3, pp. 393–422, 2012. http://dx.doi.org/10.1016/j.scico.2011.10.010

[30] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008.