DYNAMIC MATCHING AND WEAVING SEMANTICS FOR EXECUTABLE UML MODELS

Raha Ziarati

A thesis
in
The Department
of
Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science in Information System Security
Concordia University
Montréal, Québec, Canada

October 2012

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Raha Ziarati**

Entitled: **Dynamic Matching and Weaving Semantics for Executable UML Models**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Information System Security

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. Anjali Awasthi _____ Chair

Dr. Peter Grogono _____ Examiner

Dr. Benjamin Fung _____ Examiner

Dr. Mourad Debbabi _____ Supervisor

Dr. Lingyu Wang _____ Co-supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 20 _____ _____

Dr. Robin Drew, Dean

Faculty of Engineering and Computer Science

Abstract

Dynamic Matching and Weaving Semantics for Executable UML Models

Raha Ziarati

To develop more secure software, security concerns should be considered as an essential part of all phases of software development lifecycle. It has been observed that incorporation of security concerns after the completion of software development may result in conflicts between functional and security requirements and leads to severe security vulnerabilities. On the other hand, security is a crosscutting concern and consequently the integration of security solutions at the software design phase may result in scattering and tangling of security features throughout the entire design. Therefore, in the case of large scale software (e.g., hundreds of UML classes), the resulting UML design models may become more complex and difficult to understand. Moreover, adding security manually is tedious and may lead to additional security flaws.

Aspect-Oriented Modeling is an appropriate approach to systematically integrate security at the design phase as it allows the separation of crosscutting concerns from the core functionality. In this research work, we provide formal semantics for aspect matching and weaving on executable UML models, particularly for activity diagrams. The semantics is based on a defunctionalized continuation-passing style since it provides a concise and elegant description of aspect-oriented mechanisms. In addition, we have extended our framework and provided semantics for control and data flow pointcuts as these pointcuts are beneficial from a security perspective and are used to detect vulnerabilities related to information flow.

Acknowledgments

It is my pleasure to thank all those who made this thesis possible.

I must firstly thank my supervisor and mentor, Prof. Mourad Debbabi, for his endless support and guidance and patience. He helped me to become a better researcher and believed in me more than myself.

My sincere and warm thanks go to my friend and colleague, Djedjiga Mouheb, with whom I closely collaborated in my research. She was a great support and provided me invaluable assistance from the first day that I started the program.

I want to express my deepest gratitude to my parents for being on my side and supporting me all the years I was studying.

I specially want to thank my dear husband Soheil. Without his unending support, patience, and love, this thesis would not exist.

Many thanks to the following for their friendship and support: Andrei Michescu, Levon Apikian, Mariam Nouh, and Yosr Jarraya. Thanks to all those at CIISE department who helped me learn more and made my time there enjoyable.

My final debt of gratitude goes to my sister Roya to whom I owe much of who I am; I will always be thankful for your love. I dedicate this thesis to you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

Undoubtedly, software systems play a significant role in human life and are being used in different sectors from military to government to banking and healthcare. Such high reliance has resulted in the fact that huge amounts of critical and sensitive information are stored within these systems. Military secrets, bank accounts and health records are examples of them. Due to the sensitiveness of such information, security flaws can enormously impact our lives and lead to huge losses. For instance, in 2009, Albert Gonzalez and his accomplices stole more than 170 million credit/debit card numbers by hacking into the databases of retail stores [39]. They took advantage of SQL injection vulnerability [83], which is a very common flaw in web applications.

Therefore, these days security has became a necessity rather than an option in software systems. Every year, organizations across the globe spend millions of dollars on securing their software and infrastructures. Their spending on security is mostly

focused on detecting and fixing software vulnerabilities and proposing new methods to reduce risks associated with using such software. In fact, rewriting software to fix a defect may lead to fundamental modifications in the software and result in tremendous corporate expenditures in future [55]. In addition, such new modifications may also bring conflicts between functional and security requirements and produce additional security vulnerabilities.

Recent research has shown that detecting and fixing vulnerabilities as early as possible in the software development lifecycle decreases the cost of software development dramatically [5, 28, 38]. According to [38], the cost to resolve a security defect is approximately 60 times the cost of fixing the security bug in an early stage of the development. Furthermore, software that is developed with security in mind is typically more resistant against intentional attack and unintentional failures [Allen et al., 2008]. Therefore, to prevent tremendous cost growing and producing more reliable software, security concerns should be considered from the early phase of software development life cycle. In this case, vulnerabilities are remediated earlier and will not transfer from one phase to another phase.

There are two important issues in adopting such practice. First, to be able to take security into consideration in every phase of the software development, all individuals involved in entire development process need to have a sound understanding of security. Also, they should be aware of the best security solutions and recent security vulnerabilities. Unfortunately, most of the time neither a novice nor an experienced software designer (or developer) necessarily has such knowledge. As an example, in a recent State of Software Security Report [80] from one of the pioneers of secure

software development, Veracode, it is stated that most developers are in dire need of additional application security training and knowledge. They assessed 4,835 applications and more than half of them failed to meet acceptable security quality, and more than 8 out of 10 web applications failed in passing OWASP Top 10 [68]. The second issue is that security is a crosscutting concern and usually remain tangled and scattered throughout the entire software (design model or code). Therefore, in case of large scale software (e.g., hundreds of classes or million lines of code), the resulting UML design models or code may become cumbersome and hard to understand. Also, injecting security manually is tedious and generally may lead to additional security flaws.

Aspect-oriented paradigm is a promising model for addressing the previously mentioned issues. In this model, security solutions can be specified independently from applications, as general solutions, and automatically integrated into software. Therefore, without the need to be knowledgeable in security, developers can generate secure software. The usefulness of aspect-oriented techniques for enforcing security requirements in software systems has been already demonstrated in the literature [9, 59, 84, 87].

During the last decade, several Aspect-oriented Modeling (AOM) approaches have been proposed to address security concerns on UML models [32, 34, 70, 71, 88, 90]. However, in spite of the increasing interest, to date, there is neither a standard language that supports AOM, nor a standard mechanism for weaving aspects into the UML models.

Executable UML model (xUML) is a major step forward in software design phase since it enables software designers to specify models with detailed behaviors by using

action languages. In fact, the possibility of executing xUML models, allows modelers to gain a better understanding of the dynamic behaviors of their design. Additionally, since xUML models are defined in a higher level of abstraction and behaviors are specified more precisely, in-dept security concerns can be addressed in the modeling phase. For example, fixing vulnerabilities related to data flow is possible in such models, as xUML supports the assignment expression and provides actions for reading and writing variables.

There are few AOM approaches that handle xUML models [31, 40, 91] and they mainly focus on providing a framework for executing the woven model for the purposes of simulation and verification. Moreover, they are presented from a practical perspective; to date, we are not aware of any research work that explores the semantic foundations of aspect matching and weaving on xUML models.

Thus, the necessity of providing a formal semantics of aspect matching and weaving on xUML models becomes evident, as it is claimed that xUML will be the future of software modeling and such formal framework can serve as a guidelines for concrete implementations of AOM approaches with xUML supports. In addition, such semantics allow us to provide precise semantics for more security-related pointcut primitives, such as dataflow pointcut [52], which are often complex and difficult to express. Also, such semantics framework can be further used to prove some key properties or to establish some internal consistency properties.

1.2 Objectives

The main objectives of this research are as follows:

- Conduct a comparative study of the state-of-the-art research proposals in applying AOM techniques for the specification and execution of security hardening practices on software design models.
- Provide a formal semantics of aspect matching and weaving on xUML models, particularly activity diagram.
- Provide precise semantics of information flow pointcuts, which are beneficial from a security perspective as they can be used to detect a considerable number of vulnerabilities such as Cross-site scripting (XSS) [15].

1.3 Thesis Structure

The remaining of this thesis is organized as follows:

- Chapter 2 briefly presents the background related to this research topic. It introduces the UML language, Executable UML, and the aspect-oriented paradigm. Then, we describe the different techniques exists in the literature that are proposed in the literature for enforcing security on software design models. Subsequently we debate about the pros and cons of those techniques. Afterwards, we briefly reviw the concepts of λ-calculus, denotational semantics, continuations, and defunctionalization. These concepts are required for understanding the contributions of this thesis.

- Chapter 3 briefly describes an aspect-oriented modeling and weaving framework for specification and integration of security solutions into UML software design models. First, a high-level overview of the framework is presented. Then, it describes how security solutions can be specified as UML aspects. Subsequently, it presents the process of customization and integration of aspects into core UML models. Finally, a case study is provided to illustrate the approach.

- Chapter 4 provides a dynamic semantics for aspect matching and weaving for a core language based on λ-calculus. We start by presenting the syntax of the language and its denotational semantics. Then, we transform the semantics into a frame-based continuation-passing style. Afterwards, we extend the language by considering aspect-oriented constructs and provide semantics of matching and weaving. Then, we enhance our work by considering flow-based pointcuts and present an example to illustrate our proposed framework. Finally, we discuss the existing approaches that are related to our work.

- Chapter 5 provides formal semantics of aspect matching and weaving on xUML models, particularly activity diagrams. Following a similar methodology as in the previous chapter, we present the syntax of UML activity diagrams and Alf language and the associated denotational semantics. Then, we transform the semantics into continuation-passing style. Afterwards, we extend the language by considering aspect-oriented constructs and provide a semantics of matching and weaving. Afterwards, we extend the semantics with the dataflow pointcut and provide an illustrating example. Finally, we discuss the existing approaches

that are related to our work.

- Chapter 6 presents concluding remarks on our contributions. In addition, it presents a discussion of potential future research in this area.

Chapter 2

Background

In this chapter, we briefly recall the concepts that are required for understanding the contributions of this thesis. The notations that are used throughout this chapter are introduced in the appendix.

2.1 Unified Modeling Language

The Unified Modeling Language (UML) [65] is a standard modeling language, proposed by the Object Management Group (OMG), for creating an abstract model of a system, referred to as a UML model. In a few words, UML includes graphical notations used to construct and detail and document systems' artifacts. It has a four-layered architecture: (1) M0 layer, which basically contains instances (called *objects*), (2) M1 layer, which is the model of a system, (3) M2 layer, which is the model of the model (called *meta-model*), (4) M3 layer, which is the model of the meta-model (called *meta-metamodel*).

Currently, UML is at version 2.4.1 [65]. A major update has been done at version

2.0 compared to version 1.x. Providing more precise definitions of abstract syntax and semantics and a more modular language structure, and also improving capability for modeling large-scale systems are the significant enhancements of UML 2.0 over its previous version. In addition, UML now is defined in terms of the Meta Object Facility (MOF) [62], which makes it compliant with other meta-models defined by OMG. It should be noted that MOF is a modeling language that is provided for describing the elements of the M3 layer.

2.1.1 UML Diagrams

To capture different aspects of the systems, UML provides different types of diagrams. The provided diagrams can be grouped into two main categories: *structural* and *behavioral*. Table 1 and Table 2, summarize the diagrams provided in UML 2.0.

Diagrams	Description
Class Diagram	It is used to describe each individual class with its type in a system. Also, it shows that how statically classes are related to each other. This diagram is the fundamental diagram of a system design and the most frequently used UML diagram too.
Object Diagram	It is used to specify objects and their relationship at run-time.
Component Diagram	It is used to describe all component in a system, their relationships and interactions.
Composite Diagram	It is used to describe the inner structure of a component including all classes within the components and the component interfaces.
Package Diagram	It is used to structure the organization and structure of packages. Packages may contain classes or other packages within.
Deployment Diagram	It is used to describe systems hardware, software and network configurations.

Table 1: UML Structural Diagrams

Diagrams	Description
Use case Diagram	It is used to capture systems requirements and demonstrates how the systems react to requests from external users.
Activity Diagram	It is used to describe complex processes. It gives a detailed dynamic view of a specific task (process). Data flows and control flows among objects are precisely defined by using this diagram.
State Machine Diagram	It is used to describe the life cycle of objects using a finite state machine.
Sequence Diagram	It is used to describe sequence of messages passed among objects in a timeline.
Interaction Diagram	It is used to model the control flow of a system and the underlying processes.
Communication Diagram	It is used to describe the sequence of messages passed among objects in a system. It is similar to a sequence diagram, expect that it focuses more on the objects roles.
Time Sequence Diagram	It is used to describe how dynamic states change events over time. Changes may be caused by messages in state, conditions or events.

Table 2: UML Behavioral Diagrams

Structural diagrams are used to describe the static structural of elements of a system as well as relationships and dependencies between the objects. Behavioral diagrams are provided to specify the behavior of objects in a system.

2.1.2 UML Extension Mechanisms

UML is a general purpose modeling language that can be applied to all application domains. However, there are situations, in which a general language may not be suitable for modeling applications of some specific domains. To address this issue, OMG defines two possible approaches for specializing its elements, allowing customized extensions of UML for particular application domains. Introducing UML profiles which,

come as a UML Profile Package included in UML 2.0 is the first approach. The second approach is to support specifications of constraints. In the following, we provide an overview of these extension mechanisms.

UML Profiles

UML profile defines a set of UML artifacts that allows the specification of an MOF model. *Stereotypes* and *tagged values* are the main elements of the UML profile package. A *stereotype* adds new semantics and properties to existing model elements. In more details, a *stereotype* extends an existing meta element by providing additional properties (*tags*) that are specific to a particular domain. A *tag* is the name of the new property associated with a *value* which, is the actual value of that property for a given element. It is important to differentiate between class attributes and tagged values, as the value of the former applies to instances of the class while the value of the latter applies to the element itself.

UML Constraints

Constraints also extend the semantics of UML by specifying restrictions or conditions on model elements. There are some predefined constraints in UML, in addition, some constraint language are proposed to allow the specification of user-defined constraints. Object Constraint Language (OCL) [63] is a standard declarative language proposed by OMG for describing constrains on UML models. The main purposes for which OCL can be used are to: (1) query UML elements, (2) specify invariants on classes and types in the class model, (3) specify type invariants for stereotypes, (4) describe

pre and post conditions on operations, and (5) describe guards [63]. It should be noted that OCL is a pure specification language and the evaluation of OCL expressions over UML elements simply returns a value and does not change anything in the model.

2.2 Executable UML

Behaviors specified using UML diagrams are abstract and high level. In addition, no precise execution semantics is provided for all UML diagrams elements. Therefore, it is not possible for software designers to define fully executable models that can be simulated and validated before development.

The Foundational UML (fUML) [67] standard proposed by OMG to address this issue by specifying precise semantics for a subset of UML. But, the creation of executable models still remained a difficult task as the UML primitives provided in fUML are too low-level and creating reasonable sized executable UML models is close to impossible. In addition, the graphical modeling notations of UML is not always suitable for specifying detailed behaviors and it is often much more easy to do so using textual notations. To illustrate the dispute, let us consider the following example, which is taken from [54]. Figure 1 shows an activity diagram that describes the process of establishing customers' orderers. This process can be concisely expressed by few lines of Alf code as it is presented in Figure 2. Notice that the provided example describes a simple process and in the case of complex processes their corresponding activities would be much more complicated.

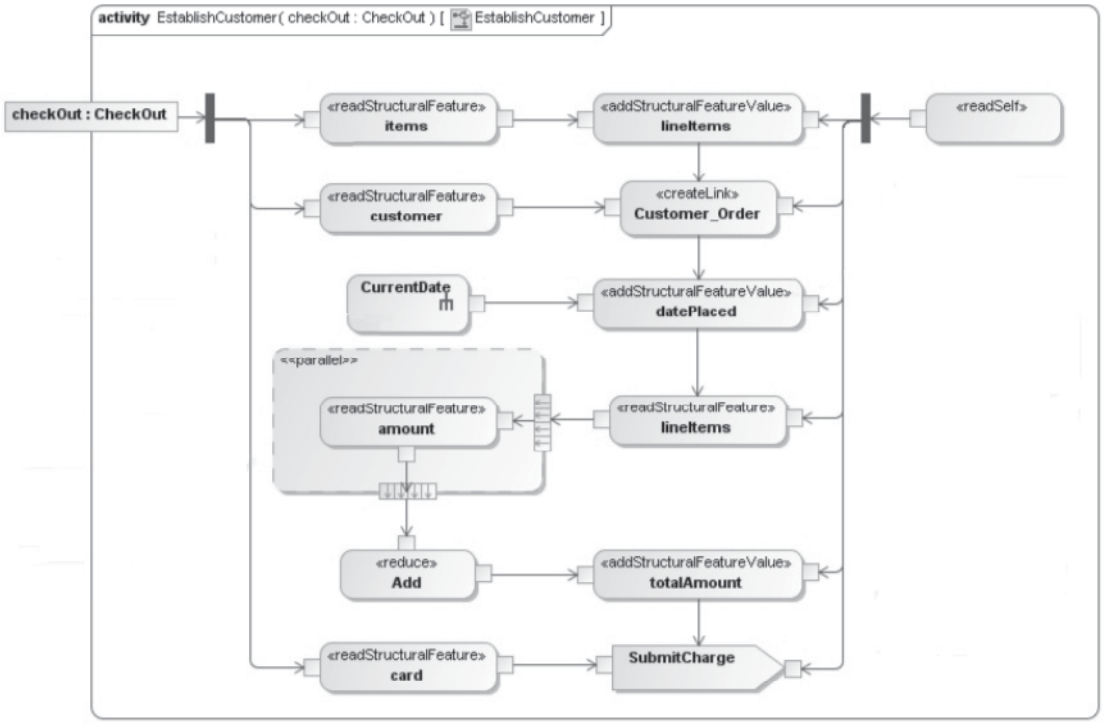

Figure 1: Establish Customer Order Activity

```
this.lineItems = checkOut.items;
Customer_Order.addLink(checkOut.customer,this);
this.datePlace = CurrentDate();
this.totalAmount = this.lineItems.amount -> reduce Add;
this.SubmitCharge (checkOut.card);
```

Figure 2: Establish Customer Order Alf Code

Therefore, for these reasons, OMG issued an RFP for a concrete syntax for an action language based on fUML. Accordingly, several UML action languages have been proposed as a high level programming languages that conform to UML action semantics such as Action Language for Foundational UML (Alf) [66], Object Action Language (OAL) [37], Action Specification Language (ASL) [49], Platform Independent Action Language (PAL) [78] , and [48]. In the following, we briefly present the

main elements of executable UML. Afterwards, we provide a brief introduction to Alf as it is the target language of this research.

2.2.1 Foundational UML

The Foundational UML (fUML) is an executable subset of standard UML that can be used to specify, in an operational style, the structural and behavioral semantics of systems. The main elements of fUML are activities, actions, structures, and asynchronous communications. In the following, we go through the basic features of activities and actions as they are used in Chapter 5. For more details, please refer to fUML specification [67].

Activities are specifications of behaviors. Nodes, edges (control/object flows), and tokens are the main elements of activities. Activity nodes mainly have three types: action nodes, object nodes, and control nodes. Actions are fundamental units of executable behaviors, which represent single steps within activities. The fUML supports various kinds of actions, which can be classified into five groups as shown below:

1. Invocations actions are provided for invoking behaviors and operations (calling activities/operations are examples of such actions)

2. Object manipulation actions are provided for objects operations (creating/destroying objects are examples of such actions)

3. Structural feature manipulation actions are provided for operations on structural features (reading/writing objects attributes are examples of such actions)

4. Association manipulation actions are provided for operations on associations. (creating/destroying objects links are examples of such actions)

5. Communication actions such as sending signals are provided for generating a synchronous form of message sending

```
result = DoSomething (1, output);
```

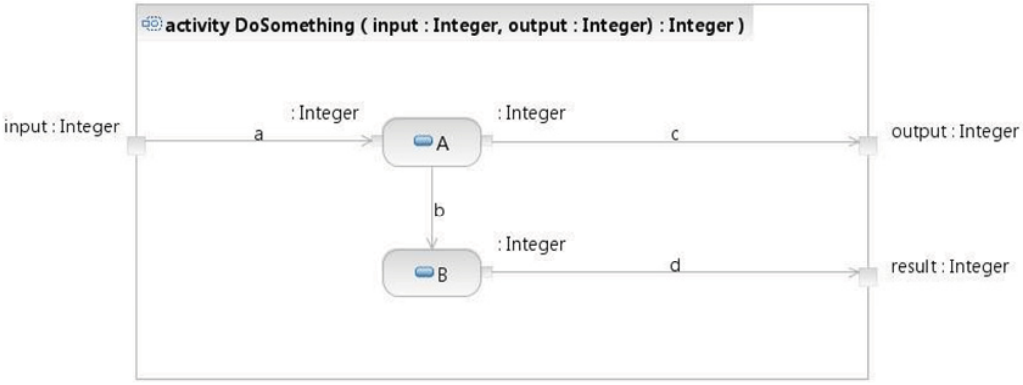

Figure 3: Example of an Activity

Figure 3 illustrates a simple activity. The activity is invoked with an argument 1 for its input parameter. Consequently an object token with a value of 1 is placed on the input activity parameter node. The object token flows to the input pin of action `A` along the object flow `a`. consequently, the action `A` fires and after the execution it produces a result as an object token and put it on its output pin. In addition, the action `A` produces a control token which follows to action `B` along the control flow b. The object flow that is produced by action `A` flows to the output activity parameter node along the object flow `c`. Meanwhile, the action `B` accepts the control token and fires, producing an object token on its output pin. The generated object token flows to the output activity parameter node along the object flow `d`.

2.2.2 Action Language for Foundational UML

Action Language for Foundational UML (Alf) is a textual surface representation for specifying executable (fUML). In addition of being a standard, Alf is highly expressive and provides the facilities required to express the actions in UML models in clear and precise and yet abstract manner. As mentioned before, the semantics of the Alf notation is defined by its mapping to fUML. Figure 4 presents the Alf notation of the activity provided in Figure 3. For extended treatment of the Alf notation, please refer to the language specifications [66].

```
activity DoSomething (in input: Integer,
                      out output: Integer): Integer
{
       output = A(input);
       return B();
}
```

Figure 4: Example of Alf Code

In the following section, we present the main concepts of aspect-oriented paradigm.

2.3 Aspect-Oriented Paradigm

The complexity of software is increasing day by day due to sophisticated functionalities required to be realized in newly developed software. During the development phase, it can be clearly observed that there are certain non-functional concerns, such as security, which have a tendency to get interleaved with the core functionalities and cannot be decomposed efficiently into single entities. Dealing with such cross-cutting concerns is a major challenge in the development of software systems as they

are scattered in various places and become inseparable from main functionalities. In this respect, Aspect-Oriented paradigm [46] is an appealing approach as it allows the encapsulation of a crosscutting concern in single unit of modularization called aspect. The main terms of aspect-oriented terminology are:

Aspect: As mentioned previously, aspects are elements that encapsulate concerns that crosscut the core functionalities of an application. Typically, an aspect consists of two elements; a set of adaptations (advice) and a set of pointcuts. Advice is a set of treatments, i.e., behaviors, which need to be injected at particular points during the execution time. These points are called join points, and the set of join points are called pointcuts.

Pointcut: A pointcut is an expression that allows the selection of a set of points in the execution flow where pieces of advice need to be injected. Each point in this set is called a join point. By analogy, a pointcut classifies join points in the same way a type classifies values.

Matching: Matching is the process of identifying join points in the execution flow.

Weaving: Weaving is the process of injecting the advice specified in the aspect at the identified join points selected by pointcuts. Figure 5 shows a high-level example of the weaving process.

The aspect-oriented paradigm originally emerged at the programming level. Many aspect-oriented programming (AOP) languages have been developed, such as, AspectJ [44] , AspectC [14], and AspectC++ [79]. However, due to the increasing interest, AOP has recently stretched over earlier stages of the software development lifecycle. Aspect-Oriented Modeling (AOM) [7] applies aspect-oriented techniques to software

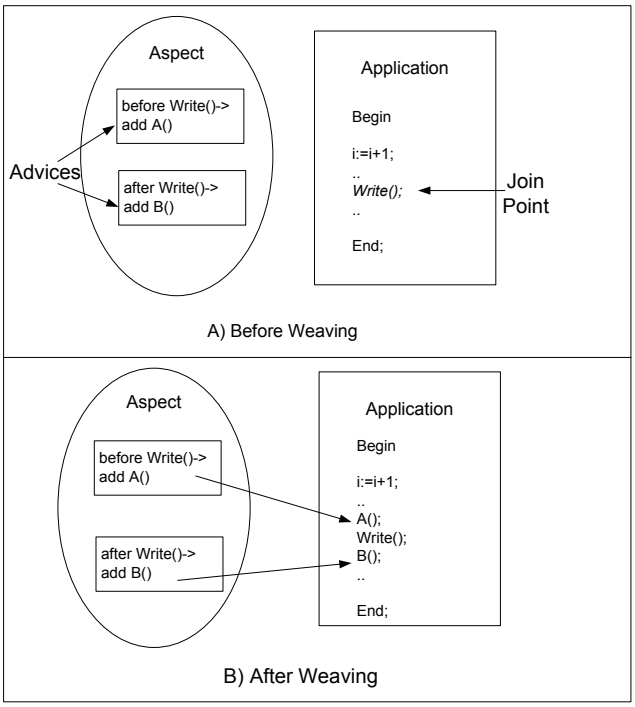

Figure 5: Example of Weaving

models with the aim of modularizing crosscutting concerns. Indeed, handling those concerns at the modeling level would significantly help in alleviating the complexity of software models and application code, as well as reducing development costs and maintenance time. In the following section, we briefly present the different methods that are proposed for security hardening on software design models.

2.4 Security Hardening on Software Design Models

Different approaches have been proposed to comfort the process of specification and integration of security solutions into software design models. Mainly, the proposed approaches can be classified into three groups; (1) security design patterns, (2) mechanism-directed meta-languages, (3) aspect-oriented modeling. In the following we briefly explain each of these categories.

2.4.1 Security Design Patterns

"A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution" [77]. Likewise, security patterns encapsulate security experts knowledge in the form of proven solutions to common problems. These solutions allow software designers to satisfy security properties of their design without having in depth knowledge about security.

Many security design patterns for security concerns such as access control, logging, cryptography, and electric signature have been proposed in the literature [8, 11, 12, 26, 42, 75, 77]. Although security design patterns provide reusable solutions to integrate security best practices during the software design phase, they have some shortcomings. Generally, the provided patterns are high-level and abstract and information about the behavior of security solutions is missed. In addition, usually structures and methodologies needed for their applications are not provided. Moreover, some of the patterns are merely textual descriptions written in English, which require manual implementation.

2.4.2 Mechanism-Directed Meta-Languages

Extension of the UML meta-language for specifying security solutions is another method that is proposed for the integration of security into design models. By providing new stereotypes and specific tagged values for defining security solutions, these approaches enable software modelers to design more secure models. In addition, UML standard extension mechanisms benefit from a good tool support since any standard

UML modeling framework supports profile specification. Many UML meta-language extensions have been proposed in the literature. The majority of them targets the specification of RBAC security policies [22, 50, 72]. Other security requirements, such as authentication, have been also addressed in [57]. In addition, mostly proposed approaches require continuous interaction between software designers and security experts in order to ensure the appropriate enforcement of security requirements.

2.4.3 Aspect-Oriented Modeling

We know that security solutions have a crosscutting nature and pervade the entire software. Also, as previously mentioned, AOM supports the idea of separating crosscutting concerns from the software core functionalities at the software modeling level. Therefore, the AOM paradigm can be a promising treatment for incorporating security at the modeling level as it enables security specialists to provide generic security solutions as aspects that can be systematically applied to design models. A security solution aspect consists of two main parts: (1) security functionalities, (2) locations where these functionalities should be applied on core software models. Concisely, when a security aspect is applied to a design model, the locations where the solution needed to be injected are identified and the functionalities specified by the aspect are injected into those locations.

Several AOM approaches have been proposed for integration of security into software models [16, 33, 34, 43, 69, 70, 71, 89, 90]. However, these approaches suffer from the lack of standardization for aspect specification and integration.

The aspect-oriented paradigm overcomes the limitations observed in the previous

approaches (see Subsection 2.4.3 and Subsection 2.4.2). By adopting aspect-oriented techniques, security experts independently specify security solutions as generic aspects. Moreover, it provides a method to automate the process of integrating security solutions into primary models. In the following section, we briefly present the main concepts of Lambda calculus.

2.5 λ-calculus

Lambda calculus (or λ-calculus) is a theory of functions introduced by Alonzo Church in the 1930s. It provides a simple notation for defining and applying functions. The notation consists of a set of λ-expressions, each of which denotes a function. A key characteristic of λ-calculus is that functions are values, just like booleans and integers. In other words, functions in λ-calculus can be passed as arguments to other functions or returned as values from other functions. First, we introduce the notations that will be used throughout this research work.

Notations

- The algorithms are written with respect to the OCaml notations [1].
- Given a record $\mathsf{D} = \{f_1 : \mathsf{D}_1; f_2 : \mathsf{D}_2; \ldots; f_n : \mathsf{D}_n\}$ and an element e of type D, the access to the field f_i of an element e is written as $e.f_i$.
- Given a record D and an element e, the notation $\mathsf{D}\ e$ in pattern matching denotes that e is of type D.
- The type $\mathsf{Identifier}$ classifies identifiers.

- Given identifiers a and b, we write $a \mapsto b$ to denote a mapping from a to b.
- Given two maps m and m', we write $m \dagger m'$ the overwriting of the map m by the associations of the map m'.

Syntax

As it is presented in [36], the pure λ-calculus contains three kinds of λ-expressions namely variables, function abstractions, and function applications. (see Figure 6)

e	$::=$	x	**variable**
	$\mid$	$\lambda x.\ e$	**abstraction**
	$\mid$	$e\ e'$	**application**

Figure 6: Syntax of λ-Calculus

1. Variables: represented by x, y, z, etc.
2. Function abstractions (or function definitions): represented by the expression $\lambda x.\ e$, where x is a variable that represents the argument and e is a λ-expression that represents the body of the function. For example: the expression $\lambda x.$ **square** x is a function abstraction that takes a variable x and returns the square of x.
3. Function applications: represented by the expression $e\ e'$, where e and e' are λ-expressions. The expression e should evaluate to a function that is then applied to the expression e'. For example, the expression $(\lambda x.$ **square** $x)\ 3$ evaluates, intuitively, to 9, which is the result of applying the squaring function $\lambda x.$ **square** x to 3. Notice that there are many strategies for evaluating function applications, and what we explained above is one way of it.

Free and Bound Variables

An occurrence of a variable in a λ-expression is either bound or free. An occurrence of a variable x in a λ-expression is bound if there is an enclosing $\lambda x.\ e$; otherwise, it is free. Let us consider the following λ-expression:

$$e = \lambda x.\ (x\ (\lambda y.\ y\ z)\ x)\ y$$

In that expression:

- Both occurrences of x are bound since they are within the scope of λx.
- The first occurrence of y is bound since it is within the scope of λy.
- The last occurrence of y is free since it is outside the scope of the λy.
- The variable z is free since there is no enclosing λz.

For extensive treatment on λ-calculus please refer to [36]. In the following section, we briefly present the main concepts of denotational semantics.

2.6 Denotational Semantics

Denotational semantics is an approach proposed by Christopher Strachey and Dana Scott in the late 1960s to provide a formal semantics of programming languages. Concisely, denotational semantics gives programs meaning (denotation) by mapping syntactic constructs of a language into mathematical objects. The important characteristic of this approach is that it is compositional and the denotation of a program built out of the denotations of its subphrases. Denotational semantics are mostly used to illustrate the essence of a language feature, without specifying how these

features are actually realized. Hence, semantics are abstract and do not provide full implementation details.

In this style, each language construct is mapped directly into its meaning by defining a semantic function $\mathcal{F}$ and a semantic domain D such that every syntactic constructs in S is mapped by $\mathcal{F}$ into elements of D, which is a structured set of abstract values such as integers, truth values, tuples of values, and functions. [56] Therefore, for each syntactic construct a semantic equation is defined to describe how the semantic function act on the construct. Figure 7 presents the denotational semantics of the λ-expressions presented in the previous section.

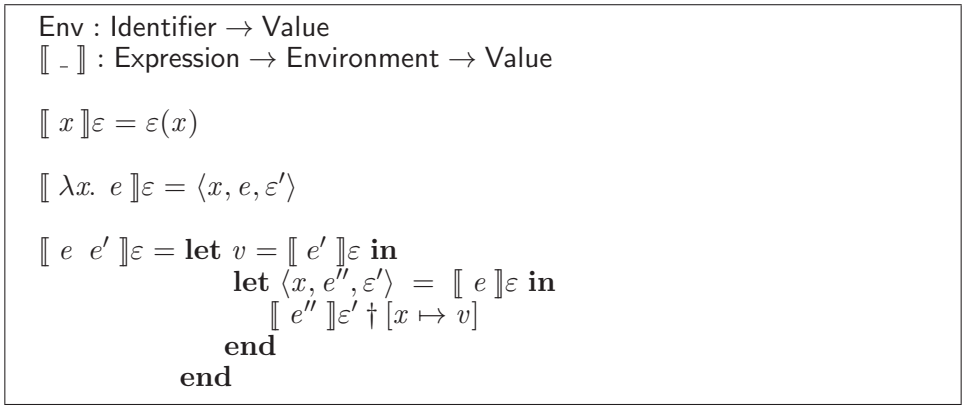

Figure 7: Denotational Semantics of λ-Calculus

Given an expression e, a dynamic environment ε, the semantic function $[\![\ \_\ ]\!]$ yields the computed value v. In the case of:

- Variables: the denotation (computed value) is the value that the variable is bound to in the environment.
- Function abstractions : the denotation is a closure $\langle x, e, \varepsilon' \rangle$ capturing the function parameter x, the function body e, and the evaluation environment ε', which

maps each free variable of e to its value at the time of the declaration of the function.

- Function applications: the denotation is computed in three steps: (1) the expression e' is evaluated, (2) the function abstraction is evaluated and the closure $\langle x, e, \varepsilon' \rangle$ is computed, (3) the expression e'' is evaluated in the environment that x is bound to the result of the evaluation of the first step.

For extensive treatment on denotational semantics please refer to [76]. In the following section, we briefly present the main concepts of continuation-passing style.

2.7 Continuation-Passing Style

Continuations first introduced in 1964 by Van Wijngaarden [74]. Later in the 1970s, many researchers [47, 73, 82] have applied the idea to the wide variety of settings [74]. In the following, we start by explaining the concept and afterward we provide the main steps of transforming direct style semantics to continuation-passing Style (CPS) semantics.

Continuations

A continuation is a function that describes the semantics of the rest of a computation. Instead of returning a value as in the familiar direct style, a function in CPS style takes another function as an additional argument to which it will pass the current computational result. This additional function argument is the continuation. To illustrate the idea of continuations, let us consider the example presented in Figure 8,

which is initially provided in [6].

```
let prodprimes n =
    if (n = 1) then 1
    else if (isprime(n)) then n * prodprimes(n − 1)
        else prodprimes(n − 1)
```

Figure 8: Function in Direct Style

The function prodprimes computes the product of all prime numbers that are less than or equal to a given number n. There are several points in the control flow of this program where control is returned. For example, the call to the function isprime returns to a point κ_1 with a boolean value b. The first call to the function prodprimes (in the **then** clause of the second **if**) returns to a point κ_2 with an integer i and the second call to prodprimes returns to a point κ_3 with an integer j. Similarly, the call to the main function prodprimes returns to a point κ with a result r. These return points represent continuations. In addition, each of these points can be considered as an additional argument to the corresponding function. When the function call terminates, this additional argument will tell us where to continue the computation. For example, the function prodprimes can be given as additional argument the return point (the continuation) κ and when it has computed its result r, it will continue by applying κ to r. The same treatment can be done to the other function calls. Figure 9 shows another version of the example presented above using continuations.

```
let prodprimes n κ =

    if (n = 1) then κ (1)

    else let κ1 b =
              if (b) then
                  let κ2 i = κ(n * i) in prodprimes(n − 1, κ2) end
              else
                  let κ3 j = κ(j) in prodprimes(n − 1, κ3) end
         in
              isprime(n, κ1)
         end
```

Figure 9: Function in CPS Style

CPS Transformation

Given a λ-expression e, it is possible to translate it into CPS. This translation is known as *CPS conversion*. In the following, we provide the main steps of this conversion:

1. Each function definition should be augmented with an additional argument; the continuation function to which it will pass the current computational result.

   let f *args* $=$ e $\Rightarrow$ **let** f *args* κ $=$ e

2. A variable or a constant in a tail position should be passed as an argument to the continuation function instead of being returned.

   return e $\Rightarrow$ κ e

3. Each function call in a tail position should be augmented with the current continuation. This is because in CPS, each function passes the result forward instead of returning it.

   return f *args* $\Rightarrow$ f *args* κ

4. Each function call, which is not in a tail position, needs to be converted into a new continuation, containing the old continuation and the rest of the computation. Here, *op* represents a primitive operation, which could include an application.

   $op\ (f\ args) \quad \Rightarrow \quad f\ args\ (\lambda r.\ \kappa\ op\ r)$

In the following section, we briefly present the main concepts of defuctionalization.

2.8 Defunctionalization

Defunctionalization proposed by Reynolds in [73], is a transformation that transforms higher-order functional programs into semantically equivalent first-order programs. The transformation consists of two main steps:

1. Represent each function abstraction to a data structure holding the free variables of the function abstraction and replace all function abstractions with their corresponding data structures.

2. Define a second-class apply function and replace all function applications with application of the apply function to a value and an argument. Basically, the apply function is a collection of the bodies of all functions and dispatches based on the type of its first argument.

Therefore, the result of the transformation is a program that contains only first-order functions. However, the original higher-order structure is implicit in the program.

Example: For a better understanding, let us consider the following example, shown in Figure 10, which is initially provided by Danvy in [19].

```
aux : (Int → Int) → Int
main : Int × Int × Bool → Int

let aux f = f 1 + f 10

let main x y b = aux(λz. z + x) * aux(λz. if b then y + z else y − z)
```

Figure 10: Higher-order Program

The function aux takes a first-class function as an argument and it applies it to 1 and 10 and outputs the summation of the applications. The main function calls aux twice and outputs the multiplication of the results. There are two function abstractions in the main function. To defunctionalize the program, we should define data types for these function abstractions and their corresponding apply function. The newly defined data types are shown in Figure 11 and their corresponding apply function is presented in Figure 12.

```
type Lam  =  Lam1 | Lam2
type Lam1 =  {id : Int}
type Lam2 =  {id : Int; cond : Bool}
```

Figure 11: New Types

```
apply : Lam × Int → Int

let apply l z = match l with
    (Lam1 l)  ⇒  l.id + z
  | (Lam2 l)  ⇒  if l.cond then l.id + z else l.id − z
```

Figure 12: Apply Function

Lastly, we rewrite the program by replacing the function abstractions with their corresponding data types and their applications with the application of the newly defined apply function. The defunctionalized program is presented in Figure 13.

$$\begin{array}{l}
\mathsf{re\text{-}aux} : \mathsf{Lam} \to \mathsf{Int} \\
\mathsf{re\text{-}main} : \mathsf{Int} \times \mathsf{Int} \times \mathsf{Bool} \to \mathsf{Int} \\
\\
\textbf{let}\ \mathsf{re\text{-}aux}\ f = \mathsf{apply}(f, 1) + \mathsf{apply}(f, 10) \\
\\
\textbf{let}\ \mathsf{re\text{-}main}\ x\ y\ b = \mathsf{re\text{-}aux}(\mathsf{Lam1}(x)) * \mathsf{re\text{-}aux}(\mathsf{Lam2}(y, b))
\end{array}$$

Figure 13: Redefined Program

Chapter 3

Static Matching and Weaving on UML Models

In this chapter, we present an aspect-oriented modeling approach for the specification and the integration of security solutions into UML software design models in a systematic manner. This approach allows software designers to focus on the main functionalities of software, and do not get diverted by non-functional requirements such as security. Later on, security solutions, which are provided by security experts, in the form of UML aspects, will be blended into the core models and target models will be generated.

I should mention that the work that is presented in this chapter is done by a team, which I was a member of it. The work had two main parts: (1) creating an AOM profile [58], and (2) implementing a model weaver [61]. I particularly was involved in the second part, which is described in subsection 3.3.3.

The remainder of this chapter is organized as follows. Section 3.1 presents a high-level overview of our AOM framework. In Section 3.2, we describe how security solutions can be specified as UML aspects. In section 3.3, we explain how an aspect is weaved into a core UML model. A case study is provided in section 3.4 to illustrate our approach. Finally, a summary of the chapter and a discussion about future work in this area are provided in Section 3.5.

3.1 Overview

A high-level overview of our approach is illustrated in Figure 14. There are two actors involved in our approach; (1) a security expert (on the right side) who is responsible for providing security solutions in form of UML aspects, and (2) software designer (on the left side) who is responsible for designing a software base model that only addresses the main functionalities of software.

The aspects designed by the security expert are generic templates representing the security features independently from the software model. In few words, each aspect consists of two main elements: security treatments that are specified as adaptations and the places where such treatments should be injected, which are designated by pointcuts. To come up with generic aspects, pointcuts are specified with parameters and are application-independent. Afterwards, by mapping the aspect parameters to elements of software base models, application-specific aspects are generated. The process of transforming an application-independent aspect to a customized and

application-specific aspect is called aspect specialization. The idea is similar to functions arguments that are unbound during the definition of the functions and will be bounds to their values when they are called.

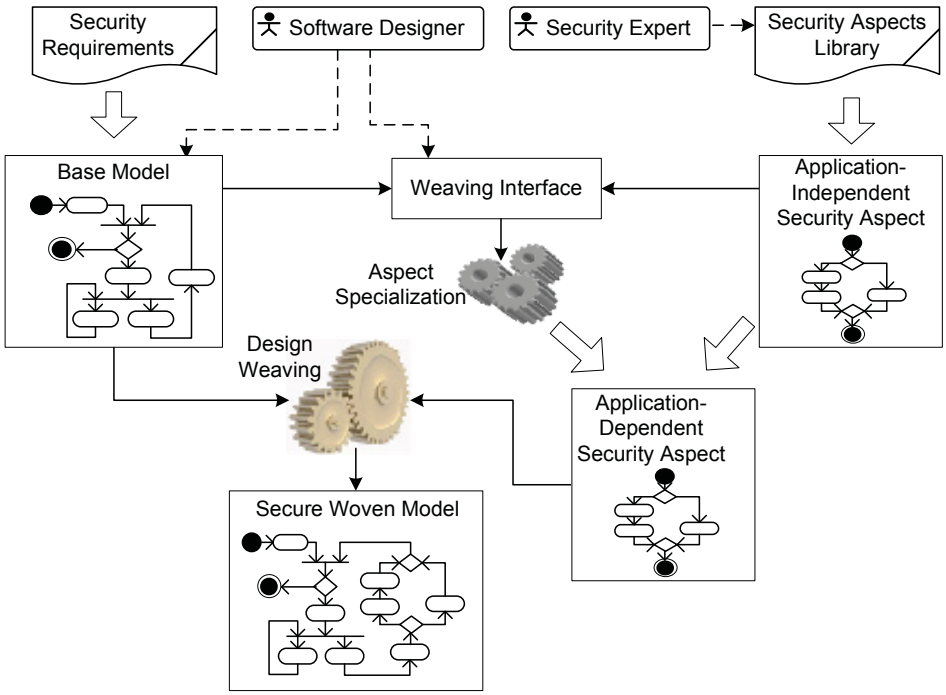

Figure 14: Overview of our Approach

Our approach works as follows. Software designers focus on the main functionalities of the software and do not get diverted by security concerns. When the base model is fully designed and created, it is the time to add the security features into the base model. Based on the security requirements specified for the software, software designers select appropriate security aspects from a security library and weave them into their base model. To facilitate the process of integration of security aspects into the model, we provide an interface that is called a weaving interface. This interface

takes both the model and the aspect as inputs and outputs a customized (application-specific) aspect. Afterwards, the weaving engine takes both the base model and the customized aspect, and automatically weaves the aspect into the model.

3.2 Aspect Specification

In order to assist security experts in designing the security aspects, an AOM profile is developed as part of our framework. This profile allows security experts to specify security solutions in the form of aspects by attaching stereotypes that are parameterized by tagged values to UML elements. The profile is designed to allow as many modification capabilities as possible.

As mentioned earlier, an aspect contains a set of adaptations and pointcuts. In our profile, an aspect is represented as a stereotyped package. In the following subsections, we show how adaptations and pointcuts can be specified using our AOM profile.

3.2.1 Adaptations

An adaptation specifies modifications that an aspect performs on the base model. Since UML allows the specification of software from multiple points of view using different types of diagrams, adaptations should also enable the specification of modifications on different types of diagrams. In our approach, we focus on those diagrams that are the most used by software designers and define two types of adaptations: structural and behavioral adaptations as depicted in Figure 15. Structural adaptations specify the modifications that affect structural diagrams and likewise behavioral

adaptations specify the modifications that affect behavioral diagrams.

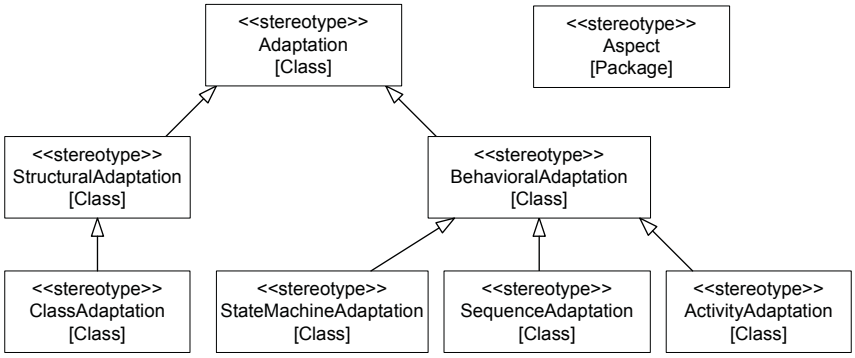

Figure 15: Meta-Language for Specifying Aspects and their Adaptations

The effects that an aspect performs on the base model elements are defined through the adaptations rules. We support two types of adaptation rules: adding a new element to the base model and removing an existing element from the base model. Figure 16 depicts our specified meta-model for adaptation rules.

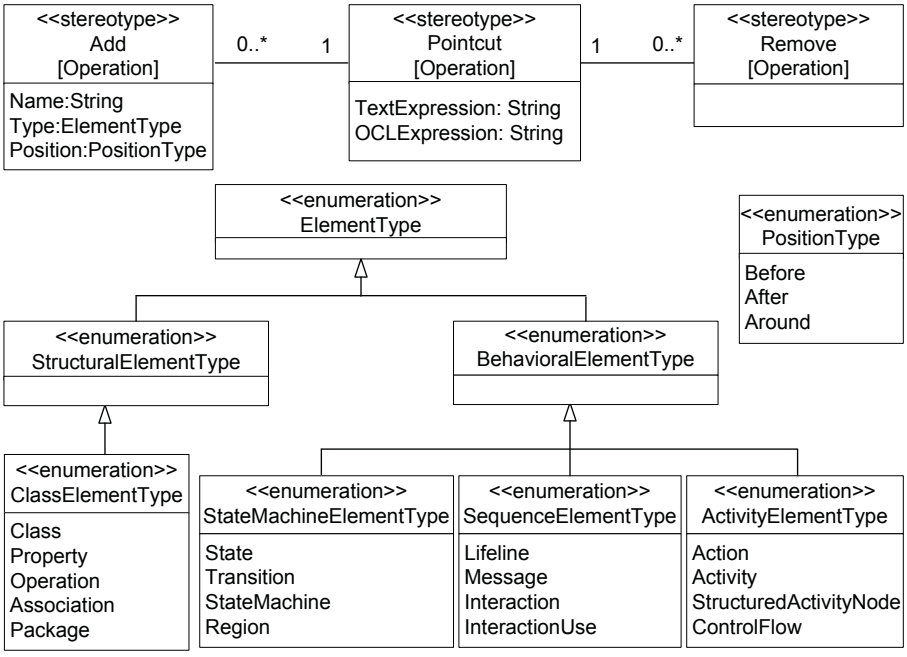

Figure 16: Meta-Language for Specifying Adaptation Rules

The addition of a new diagram element to the base model is modeled as a special kind of operation stereotyped ≪*Add*≫. Three tagged values are attached to this stereotype namely *Name*, *Type* and *Position*. The *Name* tagged value indicates the name of the element to be added to the base model. The *Type* tagged value specifies the type of the element to be added to the base model. The values of this tag are provided in the enumerations. Lastly, the *Position* tagged value that designates the position where the new element needs to be added. The values of this tag are given in the enumeration *PositionType*. This tag is needed for some elements (e.g., a message, an action) to state where exactly the new element should be added (e.g., before/after a join point). For some other elements (e.g., a class or an operation), this tag is optional. The location where the new element should be added is specified by the meta-element *Pointcut*, which will be explained in the next subsection.

The deletion of an existing element from the base model is modeled as a special kind of operation stereotyped ≪*Remove*≫. The set of elements that should be removed are given by a pointcut expression specified by the meta-element *Pointcut* (Subsection 3.2.2). Notice that, no tagged value is required for the specification of a Remove adaptation rule as the pointcut specification is enough to select the elements that should be removed.

Table 3 summarizes the main adaptation rules that are supported by our approach.

3.2.2 Pointcuts

A pointcut is an expression that allows the selection of a set of locations in the base model where adaptations should be performed. Since the targeted join points are

UML Diagram	Supported Adaptation Rules
Class Diagram	Adding/Removing a Class Adding/Removing a Property Adding/Removing an Operation Adding/Removing an Association Adding/Removing a Package
State Machine Diagram	Adding/Removing a State Machine Adding/Removing a State Adding/Removing a Transition Adding/Removing a Region
Sequence Diagram	Adding/Removing an Interaction Adding/Removing an Interaction Use Adding/Removing a Lifeline Adding/Removing a Message
Activity Diagram	Adding/Removing an Activity Adding/Removing an Action Adding/Removing a Structured Activity Node Adding/Removing a Control Flow

Table 3: Supported Adaptation Rules

UML elements, pointcuts should be defined based on designators that are specific to the UML language. To this end, we define in our approach a pointcut language that provides UML-specific pointcut designators that are needed to select UML join points. The proposed pointcut language is enough expressive to designate the main UML elements that are used in a software design. A UML element can be designated by its name, type, properties, or by its relations to other UML elements. For example, the pointcut language allows to designate a class that has a specific name and/or has its visibility property set to public. In addition, our proposed pointcut language provides high-level and user-friendly primitives that can be used intuitively by the security expert to designate UML elements.

As shown in Figure 16, the meta-element `Pointcut` is defined as stereotyped operation with two tagged values attached to it namely `TextExpression` and `OCLExpression`. The `TextExpression` tagged value is the pointcut expression specified in our proposed

textual pointcut language. An OCL expression is equivalent to the text expression, which will be generated automatically during the weaving process as we will see in the next section. The idea behind having these two tagged values is to benefit from the expressiveness of the OCL language and at the same time eliminating the overhead of writing such complex expressions by software designers. In fact, the text expression pointcut language is a high-level language, which is easy to learn and understand. However, textual expressions cannot be used to query UML elements and select the appropriate join points. Thus, in our framework, we translate the textual pointcut expressions into OCL expressions to query UML elements.

3.3 Aspect Weaving

In the previous section, we explained how we can specify UML aspects using our AOM profile. In this section, we briefly describe our proposed weaver that automatically weaves aspects that are compliant with our AOM profile into UML design models using model transformation technology.

Figure 17 presents an overview of the weaving process. As it is shown in the figure, the weaving process has three main steps: (1) aspect specialization, (2) pointcut translation, and (3) actual weaving. In the following sub-sections, we describe these three steps in more details.

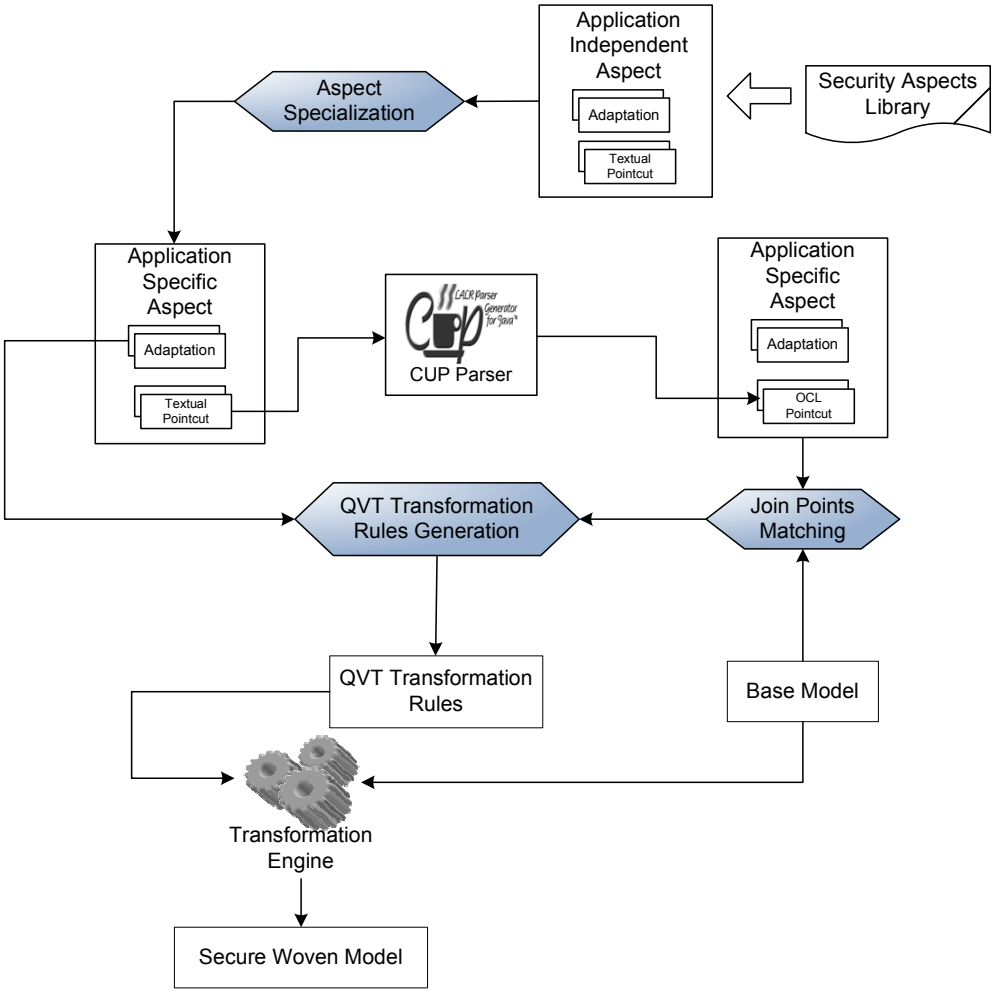

Figure 17: Overview of Weaving

3.3.1 Aspect Specialization

As we said before, the security aspects provided in the security aspect library are generic solutions. Therefore, before weaning aspects into the base models, the application-specific version of the aspects needs to be generated. This step is called aspect specialization.

To specialize an aspect, the software designer should map elements of the base model to the generic pointcuts specified in the aspect. In order to do so, we provide

a weaving interface. This interface hides the complexity of the security solutions and only exposes the generic pointcuts to the software designer. From this weaving interface and based on his/her understanding of the application, the software designer has the possibility of mapping each generic element of the aspect to its corresponding element(s) in the base model. After mapping all the generic elements, the application-specific aspect will be automatically generated.

3.3.2 Pointcut Parsing

After generating the application-specific aspect, the next step is to translate the textual pointcuts specified in the aspect to a language that can navigate through the base model and select the corresponding join points. In our approach, we chose to translate the textual pointcut expressions into the standard OCL language [63] due to the high expressiveness of the OCL language, and its conformance with UML.

This translation is done by implementing a parser that is capable of parsing and translating any textual pointcut expression, written in our high-level proposed pointcut language, to its equivalent OCL expression. Indeed, this process will be executed automatically and in a total transparent way from the user.

3.3.3 Actual Weaving

During this step, the aspect adaptations are automatically woven into the base model. In our framework, we adopt a model-to-model transformation using the standard QVT (Query/View/Transformation) language [64]. The input models of the QVT transformation are the base model and the specialized aspect model, and the generated

output model is the woven model.

Weaving is specified as a set of transformation definitions, each of which consists of a set of mapping rules. These mapping rules specify how elements of the source model should be transformed to elements in the target model. In our weaving engine, we classify the transformation definitions according to the supported UML diagrams. Thus, we provide four types of transformation definitions: class transformation definition, sequence transformation definition, activity transformation definition, and state machine transformation definition. For instance, the class transformation definition consists of a set of mapping rules, which specify how each element of the class diagram can be transformed or woven into the base model.

The actual weaving starts by parsing the adaptations specified in the aspect. Then, according to the adaptation rules, the equivalent transformation definitions will be generated. Each adaptation rule will then be translated to QVT mapping rules. These mapping rules are then interpreted by the QVT transformation engine that transforms the base model into a woven model.

3.4 Case Study

In this section, we present a case study to demonstrate the feasibility of our approach. Our case study, which is adopted from [35], presents a social networking application which has a generic login process in place. Current login process only verifies the user's username and password and either allows or blocks user access accordingly. In this scenario, a `Client` requests login page from `LoginManager` class and then calls

the `Login` method with his username and password. The `LoginManger` then validates the user credentials by contacting the `AccountManager` class. If the user credentials are valid, then the `LoginManager` requests the user's profile from `ProfileManager` class, and returns the user's homepage to the client (See Figures 18 and 19).

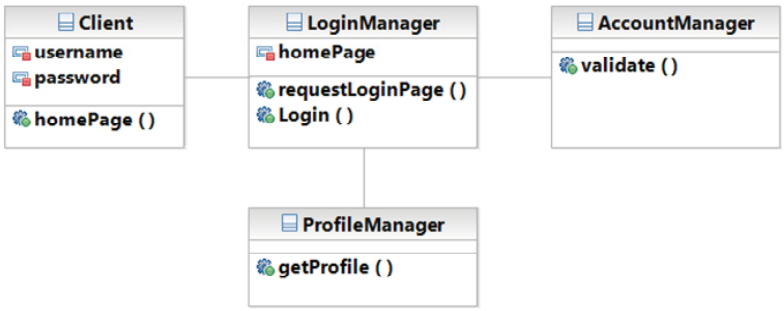

Figure 18: Class Diagram for a Social Networking Application

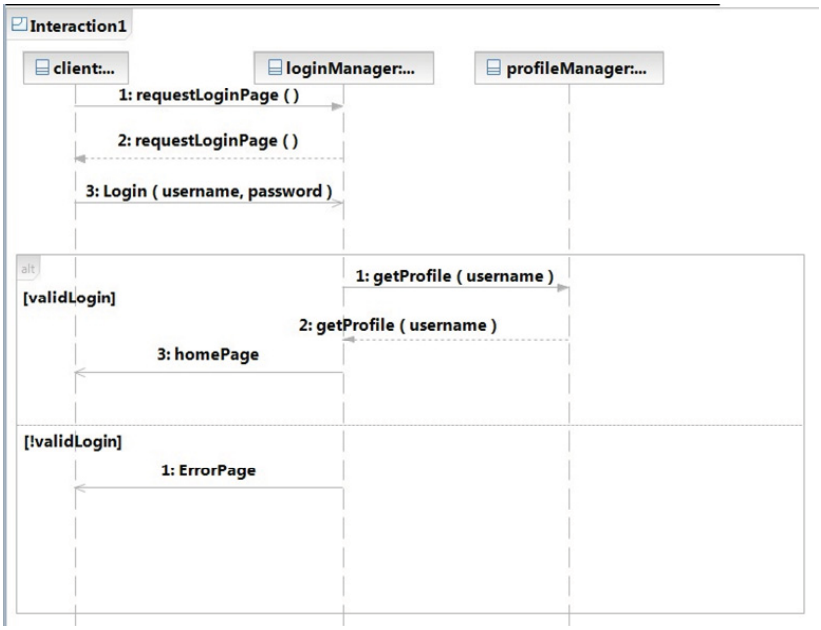

Figure 19: Sequence Diagram Representing the Login Interaction

The current login mechanism is vulnerable to different kinds of security threats,

e.g., man-in-the-middle attacks, where an attacker may intercept the user's credentials, as they are sent in plain text, and impersonate the user. To fix this problem, we replace the current login authentication mechanism with a certificate-based authentication over the Transport Layer Security (TLS) protocol [20].

To do so, we specify a TLS aspect as presented in Figure 20. The TLS aspect is designed using our AOM profile. The aspect defines two kinds of adaptations: class adaptation, and sequence adaptation. The class adaptation adds the different attributes needed by the TLS protocol, e.g., nonce, public/private keys, certificates, etc. to the `Client` and `LoginManager` classes. Additionally, it removes the current login method and replaces it with a secure one.

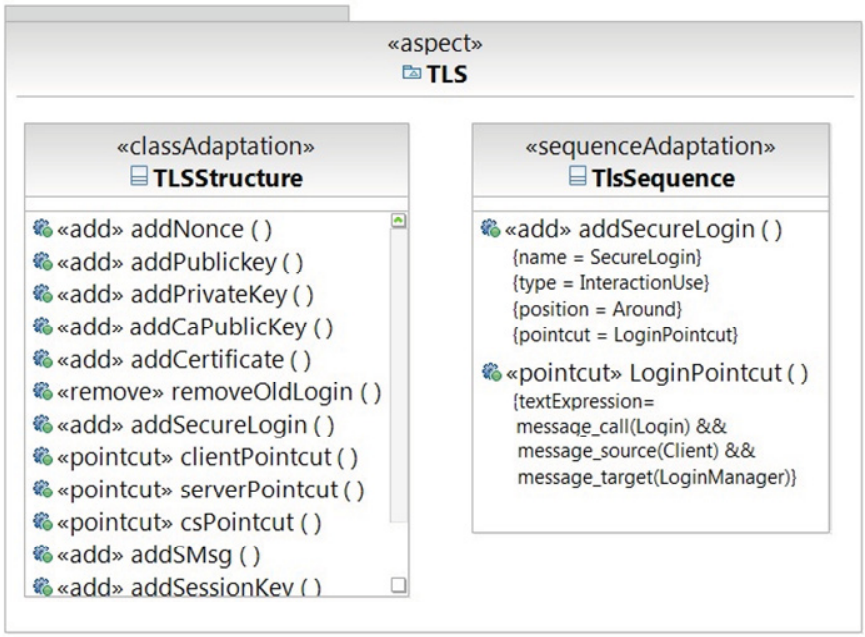

Figure 20: TLS Aspect

The sequence adaption on the other hand, adds the interaction behavior (Figure 21) that specifies the TLS protocol. This is accomplished by defining an adaptation rule `AddSecureLogin`, which specifies the injection of the secure login behavior as an interaction used around any call to the login method that is picked out by the pointcut `LoginPointcut`.

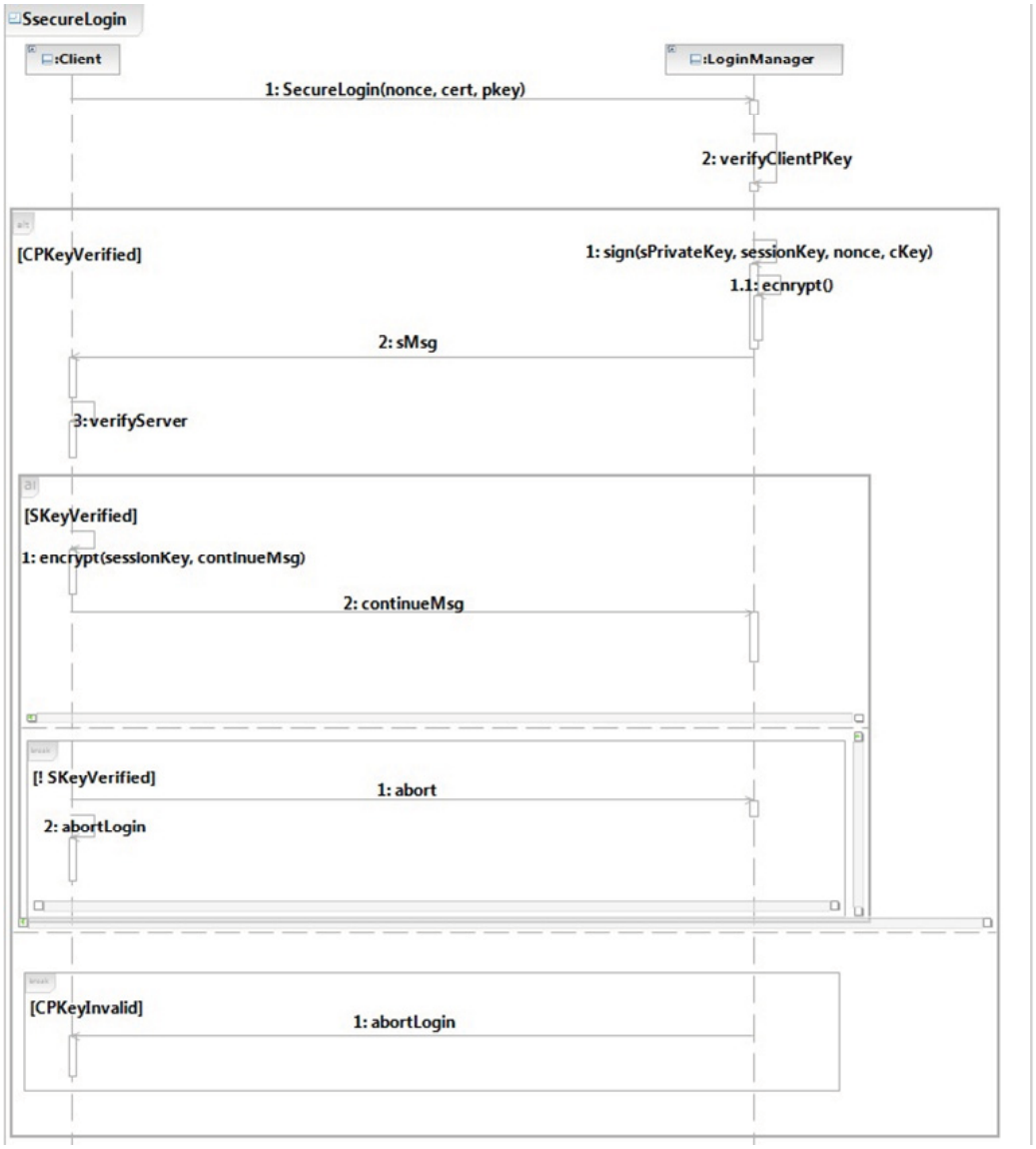

Figure 21: Sequence Diagram Representing Secure Login Interaction

The first step of the weaving is the automatic identification of the join points

where the secure login behavior (Figure 21) should be injected. To achieve this, our framework first translates the textual expression of the pointcut `LoginPointcut` to its equivalent OCL expression. This step is done automatically and in a total transparent way from the user. The resulting OCL expression is shown in Figure 22.

```
self.oclIsTypeOf(Message) and self.name= "Login" and
(self.messageSort= MessageSort::synchCall or
self.messageSort= MessageSort::asynchCall) and
self.connector._end->at(1).role.name="Client" and
self.connector._end->at(2).role.name="LoginManager"
```

Figure 22: The Resulting OCL Expression

Then, the join point matching module evaluates the generated OCL expression and returns all the message calls to the `login` operation as join points. For instance, in the example of Figure 19, the message call `Login` is selected as a matched join point. The last step of the weaving is the automatic injection of the secure login behavior into the base model at the identified join points. This is achieved by executing the QVT mapping rule that corresponds to the adaptation rule `AddSecureLogin`.

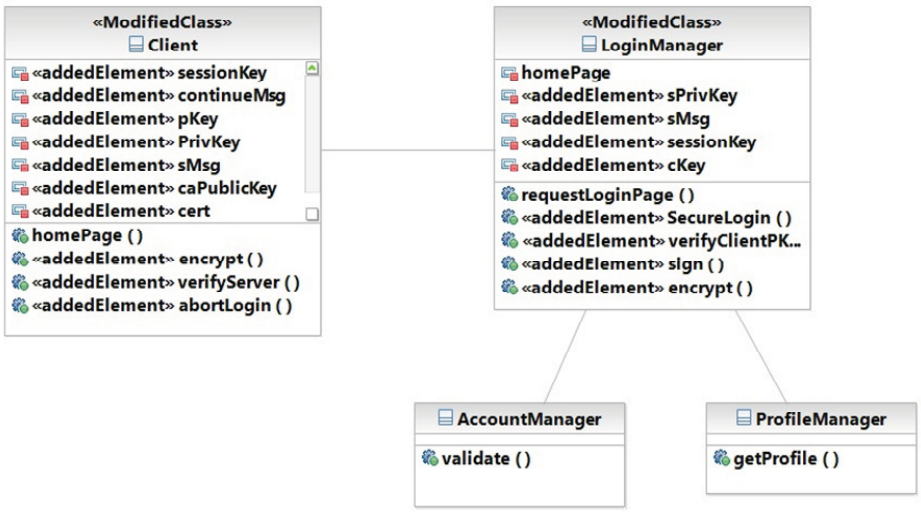

Figure 23: Social Networking Application Class Diagram

Finally, the resulting woven model is automatically generated (Figures 23and 24).

Figure 24: Sequence Diagram Representing the Secure Login Process

3.5 Summary

In this chapter, we present an AOM framework for weaving crosscutting concerns into UML models. To this end, we presented a UML profile allowing the specification of typical aspect-oriented primitives. In addition, we described a UML-specific pointcut language to designate the main UML join points. Furthermore, we elaborated a framework to specialize the generic aspects provided by security experts and automatically weave the security mechanisms into the base models. By adopting the standard OCL language for evaluating the high-level pointcuts, our approach is

generic enough to specify a wide set of pointcut expressions covering various UML diagrams. The adoption of the standard QVT language for implementing the adaptation rules extends portability of the designed weaver to all tools supporting QVT language.

We decide to enrich our framework to support executable UML models. There are two main motives for taking this decision. First, executable models enable security experts to enrich their security aspect libraries and provide aspects with more precise behaviors. Second, such models allow the security experts to provide more advanced security aspects (such as an aspect for capturing data dependencies) due to their detailed behavior specifications and execution capability.

Chapter 4

Dynamic Aspect Semantics for a Language Based on λ-calculus

Proposing a formal semantics for aspect matching and weaving on xUML models is the main objective of this research. As we mentioned before, an xUML model is a combination of UML elements and code written in an action language. Therefore, neither AOM nor AOP techniques are merely enough to perform matching and weaving on xUML models. In fact, a combination of both these techniques is needed to be able to achieve the goal.

We reach our goal by taking two steps. As the first step, we focus on the AOP side, and provide a semantics for aspects matching and weaving for a core language based on λ-calculus. We chose this language because λ-calculus serves as a basis for many programming languages, and it does not have the complexity of high-level programming languages. As the second step, we elaborate our approach to take into account model elements (the AOM side), and provide an aspect-oriented semantics

framework for xUML models. In this chapter we go through the first step and we explain the second step in the following chapter.

In our approach, we perform advice matching and weaving during the evaluation of λ-expressions. We choose Continuation-Passing Style (CPS) [81] as the basis of our semantics because, as previously demonstrated in [23], modeling aspect-oriented constructs (i.e., join points, pointcuts), in a frame-based continuation-passing style provides a concise, accurate, and elegant description of these mechanisms. Indeed, in CPS, join points arise naturally as continuation frames during the evaluation of the language expressions. In this setting, pointcuts are expressions that designate a set of continuation frames. Advice specifies actions to be performed when continuation frames satisfying a particular pointcut are activated. As it is shown in the following, by modeling join points as continuation frames, matching and weaving can be described in a simplified and unified way for different kinds of primitives and no additional structures are required to maintain the order of join points.

The remainder of this paper is organized as follows. Section 4.1 presents the syntax of a core language based on λ-calculus and its denotational semantics. We transform the semantics into a frame-based CPS style in Section 4.2. In Section 4.3, we extend the language by considering aspect-oriented constructs and subsequently we explore semantics of matching and weaving in Subsection 4.4 and Subsection 4.5 respectively. In Section 4.6, we enhance our work by considering flow-based pointcuts and present an example to illustrate the proposed framework in Subsection 4.6.3. We discuss related work in Section 4.7. Finally, a summary is presented in Section 4.8.

4.1 Syntax and Denotational Semantics

In this section, we present the syntax of the language and its denotational semantics. The notations that are used bellow are introduced in Subsection 2.5. For the sake of illustration, we choose a small core syntax that captures the essence of functional languages. The syntax is presented in Figure 25.

e	::=	c	**constant**
	|	x	**variable**
	|	$\lambda x.\ e$	**abstraction**
	|	$e\ e'$	**application**
	|	`let` x `=` e `in` e'	**local definition**
	|	`if` e_1 `then` e_2 `else` e_3	**conditional**
	|	$e_1; e_2$	**sequence**
	|	`ref` e	**referencing**
	|	`!` e	**dereferencing**
	|	e `:=` e'	**assignment**

Figure 25: The Core Syntax

We consider the following expressions:

- Constants and variables
- Functional constructs (function abstraction and function application)
- Let expressions
- Conditional expressions
- Sequential expressions
- Imperative features (referencing, dereferencing, and assignment expressions). The expression `ref` e allocates a new reference and initializes it with the value of e. The expression !e reads the value stored at the location referenced by the

value of e. The expression $e := e'$ writes the value of e' to the location referenced by the value of e.

The denotational semantics of the language is presented in Figure 26. The functions and the types used are defined as follows:

$$
\begin{array}{lcl}
\mathsf{Env} & : & \mathsf{Identifier} \to \mathsf{Value} \\
\mathsf{Store} & : & \mathsf{Location} \to \mathsf{Value} \\
\mathsf{Value} & : & \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Unit} \mid \mathsf{Location} \mid \mathsf{Closure} \\
\mathsf{Result} & : & \mathsf{Value} \times \mathsf{Store} \\
[\![\_]\!]\_\ \_ & : & \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Store} \to \mathsf{Result} \\
\mathsf{alloc} & : & \mathsf{Store} \to \mathsf{Location}
\end{array}
$$

Given an expression e, a dynamic environment ε, and a store σ, the dynamic evaluation function $[\![\_]\!]$ yields the computed value v and the updated store σ'. The environment ε maps identifiers to values. The store σ maps locations to values. A value can be either a constant, a location, or a closure. Notice that in the case of an abstraction expression $\lambda x.\ e$, the computed value is a closure $\langle x, e, \varepsilon' \rangle$ capturing the function parameter x, the function body e, and the evaluation environment ε', which maps each free variable of e to its value at the time of the declaration of the function. The function alloc used in the semantics allocates a new cell in the store and returns a reference to it.

$$[\![\ c\ ]\!]\varepsilon\ \sigma = (c, \sigma)$$

$$[\![\ x\ ]\!]\varepsilon\ \sigma = (\varepsilon(x), \sigma)$$

$$[\![\ \lambda x.\ e\ ]\!]\varepsilon\ \sigma = (\langle x, e, \varepsilon' \rangle, \sigma)$$

$$[\![\ e\ \ e'\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (v, \sigma') = [\![\ e'\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ \textbf{let}\ (\langle x, e'', \varepsilon' \rangle, \sigma'') = [\![\ e\ ]\!]\varepsilon\ \sigma'\ \textbf{in}\ [\![\ e''\ ]\!]\varepsilon' \dagger [x \mapsto v]\ \sigma''\ \textbf{end}\ \textbf{end}$$

$$[\![\ \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (v, \sigma') = [\![\ e\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ [\![\ e'\ ]\!]\varepsilon \dagger [x \mapsto v]\ \sigma'\ \textbf{end}$$

$$[\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (v, \sigma') = [\![\ e_1\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ \textbf{if}\ (v)\ \textbf{then}\ [\![\ e_2\ ]\!]\varepsilon\ \sigma'\ \textbf{else}\ [\![\ e_3\ ]\!]\varepsilon\ \sigma'\ \textbf{end}$$

$$[\![\ e_1; e_2\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (v, \sigma') = [\![\ e_1\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ [\![\ e_2\ ]\!]\varepsilon\ \sigma'\ \textbf{end}$$

$$[\![\ \texttt{ref}\ e\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (v, \sigma') = [\![\ e\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ \textbf{let}\ \ell = \mathsf{alloc}(\sigma')\ \textbf{in}\ (\ell, \sigma' \dagger [\ell \mapsto v])\ \textbf{end}\ \textbf{end}$$

$$[\![\ !\ e\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (\ell, \sigma') = [\![\ e\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ (\sigma'(\ell), \sigma')\ \textbf{end}$$

$$[\![\ e\ \texttt{:=}\ e'\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (\ell, \sigma') = [\![\ e\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ \textbf{let}\ (v, \sigma'') = [\![\ e'\ ]\!]\varepsilon\ \sigma'\ \textbf{in}\ ((), \sigma'' \dagger [\ell \mapsto v])\ \textbf{end}\ \textbf{end}$$

Figure 26: Denotational Semantics

4.2 CPS Semantics

In this section, we transform the previously defined denotational semantics into a continuation-passing style. As we mentioned earlier, frame-based semantics allows describing AOP semantics in a precise and unified way. To help understanding this transformation, we proceed in two steps. First, we elaborate a CPS semantics by representing continuations as functions. Then, we provide CPS semantics by representing continuations as frames. Continuations describe the semantics of the rest of a computation. Instead of returning a value as in the familiar direct style, a function in CPS style takes another function as an additional argument to which it will pass

the current computational result. This additional argument is called a continuation. Continuations are represented as functions, however, for the purpose of modeling join points, we need to move to a frame-based representation.

4.2.1 Function-Based Representation

The CPS semantics is presented in Figure 27. We translate the denotational semantics into CPS following the original formulation of the CPS transformation [27]. In essence, we modify the evaluation function to take a continuation as an additional argument as follows:

$$[\![\_]\!]\_\ \_\ \_ \quad : \quad \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Store} \to \mathsf{Cont} \to \mathsf{Result}$$

$$\mathsf{Cont} \quad = \quad \mathsf{Result} \to \mathsf{Result}$$

$$[\![\ c\ ]\!]\varepsilon\ \sigma\ \kappa = \kappa(c, \sigma)$$

$$[\![\ x\ ]\!]\varepsilon\ \sigma\ \kappa = \kappa(\varepsilon(x), \sigma)$$

$$[\![\ \lambda x.\ e\ ]\!]\varepsilon\ \sigma\ \kappa = \kappa(\lambda(v, \kappa').\ [\![\ e\ ]\!]\varepsilon \dagger [x \mapsto v]\ \sigma\ \kappa')$$

$$[\![\ e\ e'\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e'\ ]\!]\varepsilon\ \sigma\ (\lambda(v, \sigma').\ [\![\ e\ ]\!]\varepsilon\ \sigma'\ (\lambda f.\ f\ v\ \kappa))$$

$$[\![\ \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e\ ]\!]\varepsilon\ \sigma\ (\lambda(v, \sigma').\ [\![\ e'\ ]\!]\varepsilon \dagger [x \mapsto v]\ \sigma'\ \kappa)$$

$$[\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e_1\ ]\!]\varepsilon\ (\lambda(v, \sigma').\ \textbf{if}\ (v)\ \textbf{then}\ [\![\ e_2\ ]\!]\varepsilon\ \sigma'\ \kappa\ \textbf{else}\ [\![\ e_3\ ]\!]\varepsilon\ \sigma'\ \kappa)$$

$$[\![\ e_1; e_2\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e_1\ ]\!]\varepsilon\ \sigma\ (\lambda(v, \sigma').\ [\![\ e_2\ ]\!]\varepsilon\ \sigma'\ \kappa)$$

$$[\![\ \texttt{ref}\ e\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e\ ]\!]\varepsilon\ \sigma\ (\lambda(v, \sigma').\ \textbf{let}\ \ell = \mathsf{alloc}(\sigma')\ \textbf{in}\ \kappa(\ell, \sigma' \dagger [\ell \mapsto v])\ \ \textbf{end})$$

$$[\![\ !\ e\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e\ ]\!]\varepsilon\ \sigma\ (\lambda(\ell, \sigma').\ \kappa(\sigma'(\ell), \sigma'))$$

$$[\![\ e\ \texttt{:=}\ e'\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e\ ]\!]\varepsilon\ \sigma\ (\lambda(\ell, \sigma').\ [\![\ e'\ ]\!]\varepsilon\ \sigma'\ (\lambda(v, \sigma'').\ \kappa((), (\sigma'' \dagger [\ell \mapsto v]))))$$

Figure 27: CPS Semantics (Continuations as Functions)

The continuation, represented as a λ-expression, receives the result of the current evaluation and provides the semantics of the rest of the computation.

4.2.2 Frame-Based Representation

Continuations, which are λ-expressions, are often represented as closures. Ager *et al.* [2] have provided a systematic conversion of these closures into data structures (or frames) and an apply function interpreting the operations of those closures. This conversion is based on the concept of defunctionalization [73]. The latter is a technique by which higher-order programs, i.e., programs where functions can represent values, are transformed into first-order programs. Each frame stores the value(s) of the free variable(s) of the original continuation function and awaits the value(s) of the previous computation.

Following this technique, we transform the continuation functions obtained from the previous step into frames as shown in Figure 29. Using frame-based semantics, the continuation κ consists of a list of frames. Before presenting the semantics, we first define the primitive functions that will be used. The primitive apply, defined in Figure 28, pops the top frame from a continuation list and evaluates it based on its corresponding continuation function. When the list becomes empty, the primitive apply returns the current value and store as a result.

$$
\begin{array}{l}
\mathsf{apply} \quad : \quad \mathsf{Cont} \rightarrow (\mathsf{Value} \times \mathsf{Store}) \rightarrow (\mathsf{Value} \times \mathsf{Store}) \\
\textbf{let}\ \mathsf{apply}\ \kappa\ (v, \sigma) = \textbf{match}\ \kappa\ \textbf{with} \\
\qquad\qquad [\,] \quad \Rightarrow \quad (v, \sigma) \\
\quad \mid f :: \kappa' \quad \Rightarrow \quad \mathcal{F}[\![\, f \,]\!]\sigma\ v\ \kappa'
\end{array}
$$

Figure 28: Apply Function

The primitive push extends a continuation list with another frame.

$\mathsf{push} : \mathsf{Frame} \rightarrow \mathsf{Cont} \rightarrow \mathsf{Cont}$

$\textbf{let}\ \mathsf{push}\ f\ \kappa = f :: \kappa$

```
# GetF frame does not store any value. It awaits a location and a store.
type GetF = {}

# SetF frame stores a location. It awaits a value and a store.
type SetF = {loc : Value}

# CallF frame stores a function abstraction and an environment.
# It awaits the value of the function argument.
type CallF = {fun : Exp;  env : Env}

# ExecF frame stores the value of the argument.
# It awaits a closure, which is the result of the evaluation of the function
# abstraction and a store.
type ExecF = {arg : Value}

# LetF frame stores an identifier, a body of a let expression and an environment.
# It awaits the value of the identifier and a store.
type LetF = {id : Identifier;  exp : Exp;  env : Env}

# IfF frame stores then and else expressions and an environment.
# It awaits the value of the condition and a store.
type IfF = {thenExp : Exp;  elseExp : Exp;  env : Env}

# SeqF frame stores the next expression and an environment.
# It awaits the value of the first expression and a store.
type SeqF = {nextExp : Exp;  env : Env}

# AllocF frame does not store any value.
# It awaits the value to be stored in the newly allocated cell and a store.
type AllocF = {}

# RhsF frame stores the right-hand side expression of an assignment
# and an environment.
# It awaits a location and a store.
type RhsF = {exp : Exp;  env : Env}
```

Figure 29: Frames

In this style, the semantics is defined in two parts: the expression side (Figure 30), provides the evaluation of the language expressions, and the frame side (Figure 31), provides the evaluation of the frames that are needed for computations.

$$[\![\ c\ ]\!]\varepsilon\ \sigma\ \kappa = \ \mathsf{apply}(\kappa, (c, \sigma))$$

$$[\![\ x\ ]\!]\varepsilon\ \sigma\ \kappa = \ \mathsf{apply}(\kappa, (\varepsilon(x), \sigma))$$

$$[\![\ \lambda x.\ e\ ]\!]\varepsilon\ \sigma\ \kappa = \mathsf{apply}(\kappa, (\langle x, e, \varepsilon' \rangle, \sigma))$$

$$[\![\ e\ e'\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e'\ ]\!]\varepsilon\ \sigma\ (\mathsf{push}(\mathsf{CallF}(e, \varepsilon),\ \kappa))$$

$$[\![\ \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e\ ]\!]\varepsilon\ \sigma\ (\mathsf{push}(\mathsf{LetF}(x, e', \varepsilon), \kappa))$$

$$[\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e_1\ ]\!]\varepsilon\ \sigma\ (\mathsf{push}(\mathsf{IfF}(e_2, e_3, \varepsilon), \kappa))$$

$$[\![\ e_1; e_2\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e_1\ ]\!]\varepsilon\ \sigma\ (\mathsf{push}(\mathsf{SeqF}(e_2, \varepsilon), \kappa))$$

$$[\![\ \texttt{ref}\ e\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e\ ]\!]\varepsilon\ \sigma\ (\mathsf{push}(\mathsf{AllocF}(), \kappa))$$

$$[\![\ !\ e\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e\ ]\!]\varepsilon\ \sigma\ (\mathsf{push}(\mathsf{GetF}(), \kappa))$$

$$[\![\ e\ \texttt{:=}\ e'\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ e\ ]\!]\varepsilon\ \sigma\ (\mathsf{push}(\mathsf{RhsF}(e', \varepsilon), \kappa))$$

Figure 30: Frame-Based CPS Semantics: Expression Side

$$\mathcal{F}[\![\ \_\ ]\!]\_\ \_\ \_ \quad : \quad \mathsf{Frame} \to \mathsf{Store} \to \mathsf{Value} \to \mathsf{Cont} \to \mathsf{Result}$$

$$\mathcal{F}[\![\ \mathsf{GetF}\ f\ ]\!]\sigma\ v\ \kappa = \mathsf{apply}(\kappa, (\sigma(v), \sigma))$$

$$\mathcal{F}[\![\ \mathsf{SetF}\ f\ ]\!]\sigma\ v\ \kappa = \mathsf{apply}(\kappa, ((), \sigma \dagger [f.loc\ \mapsto v]))$$

$$\mathcal{F}[\![\ \mathsf{CallF}\ f\ ]\!]\sigma\ v\ \kappa = [\![\ f.fun\ ]\!](f.env)\ \sigma\ (\mathsf{push}(\mathsf{ExecF}(v), \kappa))$$

$$\mathcal{F}[\![\ \mathsf{ExecF}\ f\ ]\!]\sigma\ v\ \kappa = [\![\ e\ ]\!]\varepsilon' \dagger [x\ \mapsto f.arg]\ \sigma\ \kappa \quad \text{where } v = \langle x, e, \varepsilon' \rangle$$

$$\mathcal{F}[\![\ \mathsf{LetF}\ f\ ]\!]\sigma\ v\ \kappa = [\![\ f.exp\ ]\!](f.env) \dagger [f.id\ \mapsto v]\ \sigma\ \kappa$$

$$\mathcal{F}[\![\ \mathsf{IfF}\ f\ ]\!]\sigma\ v\ \kappa = \textbf{if}\ (v)\ \textbf{then}\ [\![\ f.thenExp\ ]\!](f.env)\ \sigma\ \kappa\ \ \textbf{else}\ [\![\ f.elseExp\ ]\!](f.env)\ \sigma\ \kappa$$

$$\mathcal{F}[\![\ \mathsf{SeqF}\ f\ ]\!]\sigma\ v\ \kappa = [\![\ f.nextExp\ ]\!](f.env)\ \sigma\ \kappa$$

$$\mathcal{F}[\![\ \mathsf{AllocF}\ f\ ]\!]\sigma\ v\ \kappa = \textbf{let}\ \ \ell = \mathsf{alloc}(\sigma)\ \ \textbf{in}\ \ \mathsf{apply}(\kappa, (\ell, \sigma \dagger [\ell \mapsto v]))\ \ \textbf{end}$$

$$\mathcal{F}[\![\ \mathsf{RhsF}\ f\ ]\!]\sigma\ v\ \kappa = [\![\ f.exp\ ]\!](f.env)\ \sigma\ (\mathsf{push}(\mathsf{SetF}(v), \kappa))$$

Figure 31: Frame-Based CPS Semantics: Frame Side

Example: To illustrate this transformation, let us consider the following very simple expression:

$e = (\lambda x.\ x)(1)$

By applying the CPS semantics presented in Figure 27, the expression evaluation is as follows:

$[\![\ e\ ]\!]\varepsilon\ \sigma\ \kappa = [\![\ 1\ ]\!]\varepsilon\ \sigma\ (\lambda(v, \sigma').\ [\![\ \lambda x.\ x\ ]\!]\varepsilon\ \sigma'\ (\lambda f.\ f\ v\ \kappa))$

The defunctionalization process consists of transforming the following λ-expressions into frames as shown below:

$\lambda(v, \sigma').\ [\![\ \lambda x.\ x\ ]\!]\varepsilon\ \sigma'\ (\lambda f.\ f\ v\ \kappa)$ transformed into $\mathsf{CallF}(\lambda x.\ x)$

$\lambda f.\ f\ v\ \kappa$ transformed into $\mathsf{ExecF}(1)$

Using these frames, the evaluation of the expression e is provided as follows by applying the frame semantics presented in Figure 30 and Figure 31:

$$\begin{aligned}
[\![\ e\ ]\!]\varepsilon\ \sigma\ \kappa &= [\![\ 1\ ]\!]\varepsilon\ \sigma\ (\mathsf{push}(\mathsf{CallF}(\lambda x.\ x), \kappa)) \\
&= \mathsf{apply}(\kappa, (1, \sigma)) \\
&= [\![\ \lambda x.\ x\ ]\!]\varepsilon\ \sigma\ (\mathsf{push}(\mathsf{ExecF}(1), \kappa)) \\
&= \mathsf{apply}(\kappa, (\langle x, x, \varepsilon\rangle, \sigma)) \\
&= [\![\ x\ ]\!]\varepsilon \dagger [x \mapsto 1]\ \sigma\ \kappa \\
&= \mathsf{apply}(\kappa, (\varepsilon(x), \sigma)) \\
&= (\varepsilon(x), \sigma) \\
&= (1, \sigma)
\end{aligned}$$

The frames $\mathsf{CallF}(\lambda x.\ x)$ and $\mathsf{ExecF}(1)$ correspond respectively to the moments where the function $\lambda x.\ x$ is being called and executed with an argument equal to 1. In

AOP, these moments are considered as join points where a certain advice can be applied. Thus, by transforming the denotational semantics into a frame-based style, the join points automatically arise within the semantics, which makes it an appropriate approach for defining the semantics of AOP.

4.3 Aspect Syntax and Semantics

In this section, we present our aspect extension to the language and elaborate its semantics. Our methodology in using CPS is based on a previous effort describing the semantics of a first-order procedural language (PROC) [23]. In the following, we start by presenting the aspect syntax. Then, we elaborate the matching and the weaving semantics.

An aspect, depicted in Figure 32, includes a list of advice. Advice specifies actions to be performed when join points satisfying a particular pointcut are reached.

type Aspect	=	Advice `list`
type Advice	=	$\{body$: Exp; pc : Pointcut$\}$
type Pointcut	=	GetPC | SetPC | CallPC | ExecPC | NotPC | AndPC
type GetPC	=	$\{id$: Identifier$\}$
type SetPC	=	$\{id$: Identifier; val : Value$\}$
type CallPC	=	$\{id$: Identifier; arg : Identifier$\}$
type ExecPC	=	$\{id$: Identifier; arg : Identifier$\}$
type NotPC	=	$\{pc$: Pointcut$\}$
type AndPC	=	$\{pc_1$: Pointcut; pc_2 : Pointcut$\}$

Figure 32: Aspect Syntax

Syntactically, an advice contains two parts: (1) a body, which is an expression and (2) a pointcut, which designates a set of join points. Advice can be applied before, after, or around a join point. However, before and after advice can be expressed as around advice using the `proceed` expression. Hence, we consider all kinds of advice as around advice as this does not restrict the generality of the approach.

A pointcut is an expression that designates a set of join points. We first consider the following basic pointcuts: `GetPC`, `SetPC`, `CallPC`, and `ExecPC`. The pointcut `GetPC` (resp. `SetPC`) picks out join points where the value of a variable is got from (resp. set to) the store. The pointcut `CallPC` (resp. `ExecPC`) picks out join points where a function is called (resp. executed).

As in AspectJ, advice may also compute the original join point through a special expression named `proceed` [45]. Hence, as shown in Figure 33, we extend the core syntax with an additional expression `proceed` (e) to denote the computation of the original join point with possibly a new argument e.

e	::=	...	
	|	`proceed` (e)	**proceed**

Figure 33: The `proceed` Expression

4.4 Matching Semantics

Matching is a mechanism for identifying the join points that are targeted by an advice. In a defunctionalized continuation-passing style, join points correspond to continuation frames and arise naturally when a particular continuation frame receives the value that it awaits. The matching semantics is shown in Figure 34.

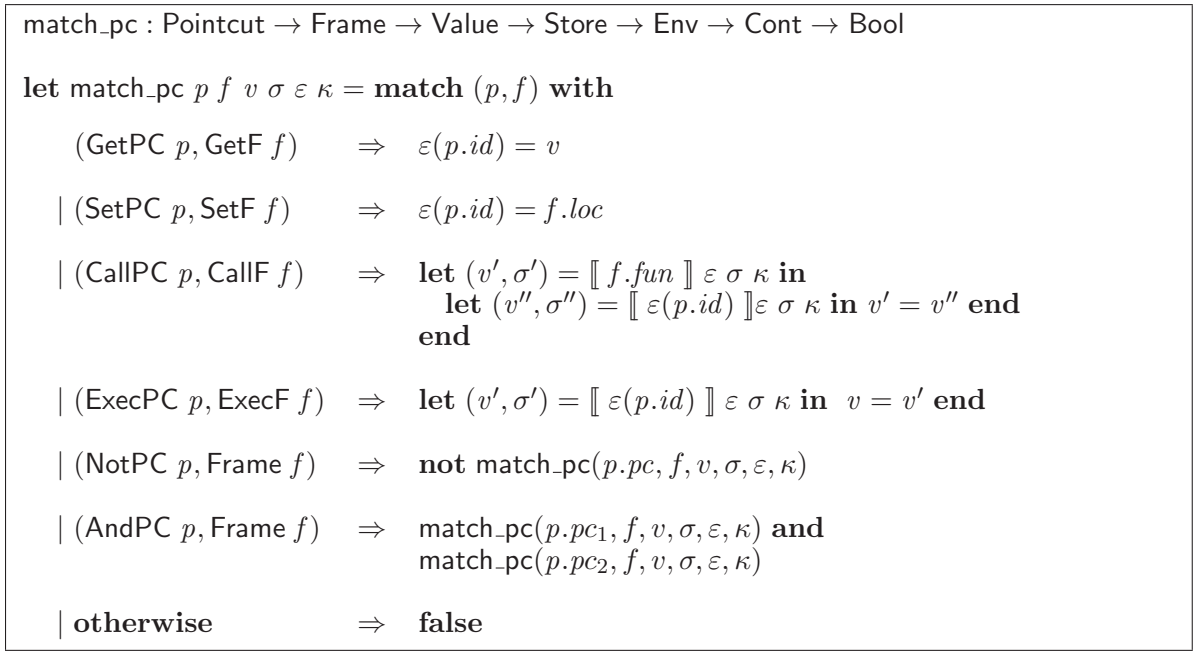

Figure 34: Matching Semantics

Given a pointcut p, the current frame f, the current value v, an environment ε, a store σ, and a continuation κ, the matching semantics examines whether f matches p. Matching depends on three factors, the kind and the content of the frame f and the current value v that f receives.

In the case of:

- GetPC pointcut, there is a match if f is a GetF frame and the location of the identifier given in p is equal to the location that f receives.

- SetPC pointcut, there is a match if f is a SetF frame and the location of the identifier given in p is equal to the location that is stored in f.

- CallPC pointcut, there is a match if f is a CallF frame and it holds a function equal to the one given in p. Notice that the pointcut p contains only the function identifier id and $\varepsilon(id)$ gives its abstraction, assuming that in the environment

identifiers map to values in case of variables, and map to function abstractions in case of functions.

- ExecPC pointcut, there is a match if f is an ExecF frame and the evaluation of the function given in p is equal to the closure that f receives.
- NotPC pointcut, there is a match if f does not match the sub-pointcut of p. (The sub-pointcut of pointcut p is the pointcut, which is enclosed in p)
- AndPC pointcut, there is a match if f matches both its sub-pointcuts.

Example: Let us consider the previous expression (slightly changed to define a function f):

$e = (\textbf{let } f = \lambda x.\ x \textbf{ in } f(1) \textbf{ end})$

and a pointcut p that captures any call to the function f with an argument x:

CallPC $p = \{id = f;\ arg = x\}$

As shown in the previous section, the frame-based semantics of the expression e usees the frames CallF$(\lambda x.\ x)$ and ExecF(1), which correspond to the moments where the function $\lambda x.\ x$ is called and executed respectively. By applying the matching semantics presented in Figure 34, it is clear that the pointcut p matches the frame CallF$(\lambda x.\ x)$.

4.5 Weaving Semantics

The weaving semantics describes how to apply the matching advice at the identified join points. Since join points correspond to continuation frames, the advice body

provides a means to modify the behavior of those continuation frames. The weaving is performed directly in the evaluation function. To do so, we redefine the apply function, as shown in Figure 35, to take an aspect α and an environment ε into account. Accordingly, the signatures of the evaluation functions as well as the matching one are also modified to take the aspect and the environment as additional arguments.

```
apply : Cont → (Value × Store) → Env → Aspect → (Value × Store)

let apply κ (v, σ) ε α = match κ with

          [ ]   ⇒   (v, σ)

    | f :: κ'   ⇒   let ms = get_matches(f, v, σ, ε, α, κ') in
                      if ms = [ ] then F⟦ f ⟧ε σ v α κ'
                      else
                        let argV = match f with
                            SetF f        ⇒ v
                          | CallF f       ⇒ v
                          | ExecF f       ⇒ f.arg
                          | otherwise     ⇒ ()
                        in execute_advice(ms, f, argV, σ, ε, α, κ')
                        end
                    end
```

Figure 35: Redefined Apply Function

The weaving is done in two steps. When a continuation frame is activated, we first check for a matching advice by calling the get_matches function. If there is any applicable advice, the function execute_advice is called. Otherwise, the original computation is performed. In the following, we explain these two steps.

Advice Matching

Advice matching is shown in Figure 36. To get applicable advice, we go through the aspect and check whether their enclosed pointcuts match the current frame. This is done by using the function match_pc defined previously in Figure 34. In case there

is a match, we return a structure MatchedAD containing the advice itself and the pointcut arguments that will pass values to the advice execution.

```
type MatchedAD   =   {arg : Identifier; ad : Advice}
get_matches      :   Frame → Value → Store → Env → Aspect → Cont →
                     MatchedAD list

let get_matches f v σ ε α κ = match α with
        [ ]   ⇒   [ ]

  | ad :: α'   ⇒   let p = ad.pc in
                     if match_pc(p, f, v, σ, ε, α, κ) then
                       let arg = match p with
                           SetPC p              ⇒ p.id
                         | CallPC p | ExecPC p ⇒ p.arg
                         | otherwise           ⇒ ()
                       in MatchedAD(arg, ad) :: get_matches(f, v, σ, ε, α', κ)
                       end
                     else get_matches(f, v, σ, ε, α', κ)
                   end
```

Figure 36: Advice Matching

Advice Execution

Advice execution is shown in Figure 37. It starts by evaluating the body of the first applicable advice. The remaining applicable pieces of advice as well as the current frame are stored in the environment by binding them to auxiliary variables &*proceed* and &*jp* respectively. To evaluate the advice body, we define a new continuation frame, AdvExecF, as follows:

$$\textbf{type}\ \mathsf{AdvExecF} = \{matches : \mathsf{MatchedAD}\ \texttt{list};\ jp : \mathsf{Frame}\}$$

$$\mathcal{F}[\![\ \mathsf{AdvExecF}\ f\ ]\!]\varepsilon\ \sigma\ v\ \alpha\ \kappa = \ \mathsf{execute\_advice}(f.matches, f.jp, v, \sigma, \varepsilon, \alpha, \kappa)$$

The evaluation of the `proceed` expression is provided below. The value of its argument is passed to the next advice or to the current join point if there is no

execute_advice :
MatchedAD list → Frame → Value → Store → Env → Aspect → Cont → Result

let execute_advice $ms\ f\ v\ \sigma\ \varepsilon\ \alpha\ \kappa =$ **match** ms **with**

$$\begin{array}{rcl} [\,] & \Rightarrow & \mathsf{apply}(\mathsf{push}(\mathsf{MarkerF}(), (\mathsf{push}(f, \kappa))), (v, \sigma), \varepsilon, \alpha) \\ \mid m :: ms' & \Rightarrow & \textbf{let } ad = m.ad \textbf{ in} \\ & & \quad [\![\ ad.body\ ]\!]\varepsilon \dagger [\&proceed \mapsto ms', \&jp \mapsto f, m.arg \mapsto v]\ \sigma\ \alpha\ \kappa \\ & & \textbf{end} \end{array}$$

Figure 37: Advice Execution

further advice. To execute the remaining pieces of advice, the AdvExecF frame is added to the list of frames.

$$[\![\ \texttt{proceed}\ (e)\ ]\!]\varepsilon\ \sigma\ \alpha\ \kappa = [\![\ e\ ]\!]\varepsilon\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{AdvExecF}(\varepsilon(\&proceed), \varepsilon(\&jp)), \kappa))$$

When all applicable pieces of advice are executed, the original computation, i.e., the current join point is invoked. To avoid matching the currently matched frame repeatedly, we introduce a new frame, MarkerF, which invokes the primary apply function, renamed here as apply_prim.

$$\textbf{type } \mathsf{MarkerF} = \{\ \}$$

$$\mathcal{F}[\![\ \mathsf{MarkerF}\ f\ ]\!]\varepsilon\ \sigma\ v\ \alpha\ \kappa = \ \mathsf{apply\_prim}(\kappa, (v, \sigma))$$

Example: If we consider the previous example:

Expression: $e = (\textbf{let } f = \lambda x.\ x \textbf{ in } f(1) \textbf{ end})$

Pointcut: $\mathsf{CallPC}\ p = \{id = f;\ arg = x\}$

and we define the advice a as:

$$\mathsf{Advice}\ a = \{body = \texttt{proceed}\ (2);\quad pc = p\}$$

As we have seen in the matching semantics, the $\mathsf{CallF}(\lambda x.\ x)$ frame is matched as a

join point. Advice a is then executed at the moment when this frame is extracted from the continuation list, i.e., when it receives the value of the argument. Since advice body is `proceed` (2), the frame $\mathsf{CallF}(\lambda x.\ x)$ will be evaluated with an argument equal to 2 instead of 1.

4.6 Flow-Based Pointcuts Semantics

In this section, we extend our framework by considering flow-based pointcuts, namely, control flow (`cflow`) and dataflow (`dflow`) pointcuts. These pointcuts are useful from a security perspective since they can detect a considerable number of vulnerabilities related to information flow, such as Cross-site Scripting (XSS) and SQL injection attacks [29]. First, we extend the aspect syntax with these two pointcuts as shown in Figure 38 and then we provide their semantics in the following sub-sections.

type Pointcut	=	... | CFlowPC | DFlowPC
type CFlowPC	=	$\{pc$: Pointcut$\}$
type DFlowPC	=	$\{pc$: Pointcut; tag : Identifier$\}$

Figure 38: Syntax of `cflow` and `dflow` Pointcuts

4.6.1 Control-Flow Pointcut

The control flow pointcut, `cflow`*(p)*, picks out each join point in the control flow of the join points picked out by the pointcut p [45]. One of the techniques that are used to implement `cflow` is the stack-based approach [21, 53]. The latter maintains a stack of join points. The algorithm for matching a `cflow` pointcut starts from the

top of the stack and matches each join point against p. If there is a match then the current join point satisfies the `cflow` pointcut [53]. Implementing the `cflow` pointcut by adopting this approach in our framework is straightforward as the stack of join points corresponds to the list of continuation frames in our model. Figure 39 shows the `cflow` matching semantics.

```
type JpF = GetF | SetF | CallF | ExecF

let match_pc p f v σ ε α κ = match (p, f) with
  ...
 | (CFlowPC p, JpF f)   ⇒   let b1 = match_pc(p.pc, f, v, σ, ε, α, κ) in
                              if (b1) then
                                let κ' = push(CflowF(p.pc), κ) in b1 end
                              else
                                exists(CflowF(p.pc), κ)
                            end
```

Figure 39: Matching Semantics of the `cflow` Pointcut

When a frame matches the sub-pointcut p of a `cflow` pointcut, a special marker frame, CFlowF, is pushed into the continuation list. The purpose of using this marker frame is to detect exit points of join points that match p. For example, if p is a `call` pointcut, the marker frame is pushed into the continuation list if the top frame matches p. Then, the marker frame will be popped when the evaluation of the function call terminates. The CFlowF is defined as follows:

type CFlowF = $\{pc : \text{Pointcut}\}$

$\mathcal{F}[\![\ \text{CFlowF}\ f\ ]\!]\varepsilon\ \sigma\ v\ \alpha\ \kappa = \ \text{apply}(\kappa, (v, \sigma), \varepsilon, \alpha)$

In summary, a join point frame `f` matches a `cflow` pointcut that contains a pointcut p if: (1) f matches the sub-pointcut p, or (2) a CFlowF marker frame that contains p exists in the continuation list. The primitive function exists used in the matching

semantics is defined in Figure 40. This function takes a frame f and a continuation list κ and checks whether f exists in the list or not.

```
exists : Frame → Cont → Bool

let exists f κ = match κ with
        [ ]          ⇒   false

      | f' :: κ'     ⇒   let b = match f' with
                               CflowF f'    ⇒ f'.pc = f.pc
                             | otherwise    ⇒ false
                         in b or exists(f, κ')
                         end
```

Figure 40: Exists Function

4.6.2 Data-Flow Pointcut

The dataflow pointcut, as defined in [52], picks out join points based on the origins of values, i.e., `dflow`$[x, x'](p)$ matches a join point if the value of x originates from the value of x'. Variable x should be bound to a value in the current join point whereas variable x' should be bound to a value in a past join point matched by p. Therefore, `dflow` must be used in conjunction with some other pointcut that binds x to a value in the current join point [52].

To match `dflow` pointcuts, particular tags are assigned to the `dflow` pointcuts to discriminate `dflow` pointcuts and track dependencies between values [52]. Briefly, if an expression matches the sub-pointcut of a `dflow` pointcut, p, this expression is tagged with the tag of this `dflow` pointcut. This tag is then propagated to other expressions that are data-dependent on the expression that matches the sub-pointcut. The `dflow` pointcut is useful where information flow is important, such as to detect

input validation vulnerabilities in Web applications.

As defined in Figure 38, the `dflow` pointcut has a sub-pointcut pc and a unique tag that discriminates this `dflow` pointcut from other `dflow` pointcuts. In order to track dependencies between values, we use a tagging environment γ that maps values to tags. As shown in Figures 41 and 42, tag propagation is performed dynamically at the same time we evaluate expressions. Thus, we augment the signatures of the evaluation functions as well as the apply function with the tagging environment as follows:

$$
\begin{array}{lcl}
[\![\_]\!]\_\_\_\_\_ & : & \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Tag\_Env} \to \mathsf{Store} \to \mathsf{Aspect} \to \mathsf{Cont} \\
 & & \to \mathsf{Result} \\
\mathcal{F}[\![\_]\!]\_\_\_\_\_\_ & : & \mathsf{Frame} \to \mathsf{Env} \to \mathsf{Tag\_Env} \to \mathsf{Store} \to \mathsf{Value} \to \mathsf{Aspect} \\
 & & \to \mathsf{Cont} \to \mathsf{Result} \\
\mathsf{apply} & : & \mathsf{Cont} \to (\mathsf{Value} \times \mathsf{Store}) \to \mathsf{Env} \to \mathsf{Tag\_Env} \to \mathsf{Aspect} \\
 & & \to (\mathsf{Value} \times \mathsf{Store})
\end{array}
$$

Notice that the definition of the apply function (see Figure 35) does not change. Only the tagging environment is passed to the matching function. Notice also that in the case of an abstraction expression, the closure $\langle x, e, \varepsilon' \rangle$ is extended with a tagging environment γ' to capture the tags generated during the execution of the function. In addition, we define a marker frame DflowF that is used for tag propagation in the case of an application expression. The frame stores a tagging environment before entering a function call and awaits the result of the call.

type $\mathsf{DflowF} = \{tag\_env : \mathsf{Env}\}$

$$[\![\ c\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = \ \mathsf{apply}(\kappa, (c, \sigma), \varepsilon, \gamma \dagger [c \mapsto \{\ \}], \alpha)$$

$$[\![\ x\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = \ \mathsf{apply}(\kappa, (\varepsilon(x), \sigma), \varepsilon, \gamma, \alpha)$$

$$[\![\ \lambda x.\ e\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = \mathsf{apply}(\kappa, (\langle x, e, \varepsilon', \gamma' \rangle, \sigma), \varepsilon, \gamma, \alpha)$$

$$[\![\ e\ e'\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = [\![\ e'\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{CallF}(e, \varepsilon),\ \kappa))$$

$$[\![\ \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = [\![\ e\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{LetF}(x, e', \varepsilon), \kappa))$$

$$[\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = [\![\ e_1\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{IfF}(e_2, e_3, \varepsilon), \kappa))$$

$$[\![\ e_1; e_2\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = [\![\ e_1\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{SeqF}(e_2, \varepsilon), \kappa))$$

$$[\![\ \texttt{ref}\ e\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = [\![\ e\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{AllocF}(), \kappa))$$

$$[\![\ !\ e\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = [\![\ e\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{GetF}(), \kappa))$$

$$[\![\ e\ \texttt{:=}\ e'\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = [\![\ e\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{RhsF}(e', \varepsilon), \kappa))$$

$$[\![\ \texttt{proceed}\ (e)\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ \kappa = [\![\ e\ ]\!]\varepsilon\ \gamma\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{AdvExecF}(\varepsilon(\&proceed), \varepsilon(\&jp)), \kappa))$$

Figure 41: Frame-Based CPS Semantics with the `dflow` Pointcut: Expression Side

In the following, we explain the tag propagation rules for the affected expressions:

- The value of a constant is associated with an empty set.
- In the case of an application expression $e\ e'$, the tags of the value of the argument e' propagate to the value of the variable x. This is performed during the evaluation of the `ExecF` frame as shown in Figure 42. In addition, the tags of the argument as well as the tags that are generated during the execution of the function body propagate to the result of the function call. For this reason, we use a DflowF frame to access the result of the function call and restore the tagging environment after returning from the call. The function $\mathsf{getTags}(\gamma)$ is used to retrieve all the tags stored in the tagging environment γ.
- In the case of a `let` expression ($\texttt{let}\ x = e\ \texttt{in}\ e'$), the tags of the value of the

$$\mathcal{F}[\![\ \mathsf{GetF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathsf{apply}(\kappa, (\sigma(v), \sigma), \varepsilon, \gamma \dagger [\sigma(v) \mapsto \gamma(v)], \alpha)$$

$$\mathcal{F}[\![\ \mathsf{SetF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathsf{apply}(\kappa, ((), \sigma \dagger [f.loc\ \mapsto v]), \varepsilon, \gamma \dagger [f.loc\ \mapsto \gamma(v)], \alpha)$$

$$\mathcal{F}[\![\ \mathsf{CallF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = [\![\ f.fun\ ]\!](f.env)\ \gamma\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{ExecF}(v), \kappa))$$

$$\begin{aligned}\mathcal{F}[\![\ \mathsf{ExecF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = {} & [\![\ e\ ]\!](\varepsilon' \dagger [x\ \mapsto f.arg]) \\ & (\gamma' \dagger [\varepsilon(x) \mapsto \gamma(f.arg)])\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{DflowF}(\gamma), \kappa)) \\ & \text{where } v = \langle x, e, \varepsilon', \gamma' \rangle\end{aligned}$$

$$\mathcal{F}[\![\ \mathsf{LetF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = [\![\ f.exp\ ]\!](f.env \dagger [f.id\ \mapsto v])(\gamma \dagger [\varepsilon(f.id) \mapsto \gamma(v)])\ \sigma\ \kappa$$

$$\begin{aligned}\mathcal{F}[\![\ \mathsf{IfF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = {} & \mathbf{if}\ (v)\ \mathbf{then}\ [\![\ f.thenExp\ ]\!](f.env)\ \gamma\ \sigma\ \alpha\ \kappa \\ & \mathbf{else}\ [\![\ f.elseExp\ ]\!](f.env)\ \gamma\ \sigma\ \alpha\ \kappa\end{aligned}$$

$$\mathcal{F}[\![\ \mathsf{SeqF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = [\![\ f.nextExp\ ]\!](f.env)\ \gamma\ \sigma\ \alpha\ \kappa$$

$$\begin{aligned}\mathcal{F}[\![\ \mathsf{AllocF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = {} & \mathbf{let}\ \ \ell = \mathsf{alloc}(\sigma)\ \ \mathbf{in} \\ & \quad \mathsf{apply}(\kappa, (\ell, \sigma \dagger [\ell \mapsto v]), \varepsilon, \gamma \dagger [\ell \mapsto \gamma(v)], \alpha) \\ & \mathbf{end}\end{aligned}$$

$$\mathcal{F}[\![\ \mathsf{RhsF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = [\![\ f.exp\ ]\!](f.env)\ \gamma\ \sigma\ \alpha\ (\mathsf{push}(\mathsf{SetF}(v), \kappa))$$

$$\mathcal{F}[\![\ \mathsf{AdvExecF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathsf{execute\_advice}(f.matches, f.jp, v, \sigma, \varepsilon, \gamma, \alpha, \kappa)$$

$$\mathcal{F}[\![\ \mathsf{MarkerF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathsf{apply\_prim}(\kappa, (v, \sigma))$$

$$\mathcal{F}[\![\ \mathsf{CFlowF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathsf{apply}(\kappa, (v, \sigma), \varepsilon, \gamma, \alpha)$$

$$\mathcal{F}[\![\ \mathsf{DFlowF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathsf{apply}(\kappa, (v, \sigma), \varepsilon, f.tag\_env \dagger [v \mapsto \mathsf{getTags}(\gamma)], \alpha)$$

Figure 42: Frame-Based CPS Semantics with the `dflow` Pointcut: Frame Side

expression e propagate to the value of x. This is performed during the evaluation of the `LetF` frame as shown in Figure 42.

- In the case of a referencing expression `ref` e, the tags of the value of the expression e propagate to the value of the expression `ref` e. This is performed during the evaluation of the `AllocF` frame as shown in Figure 42.

- In the case of a dereferencing expression `!`e, the tags of the value of the reference e propagate to the value stored at that reference. This is performed during the

evaluation of the `GetF` frame as shown in Figure 42.

- In the case of an assignment expression $e := e'$, the tags of the value of the expression e' propagate to the value of the expression e. This is performed during the evaluation of the `SetF` frame as shown in Figure 42.

The matching semantics of the `dflow` pointcut is presented in Figure 43. A join point frame f matches a `dflow` pointcut that contains a pointcut pc and a tag t if: (1) the frame f matches the pointcut pc of the `dflow` pointcut, or (2) the set of tags of the value that the frame f awaits (captured by the variable val') contains the tag t. In case a frame f matches the pointcut pc of the `dflow` pointcut, the tag t propagates to the value associated with the frame f (captured by the variable val).

```
let match_pc p f v σ ε γ α κ = match (p, f) with
  ...
  | (DFlowPC p, JpF f)   ⇒   let (b, γ') = match_pc(p.pc, f, v, σ, ε, γ, α, κ) in
                               let val = match f with
                                 GetF f     ⇒ v
                                 SetF f     ⇒ v
                                 CallF f    ⇒ let p = p.pc in
                                                let (v', σ') =
                                                   ⟦ ε(p.id) ⟧ε γ σ α κ in
                                                v'
                                              end
                                            end
                                 ExecF f    ⇒ v
                               in
                                 if (b)
                                 then (true, γ' † [val ↦ γ'(val) ∪ {p.tag}])
                                 else let val' = match f with
                                                   CallF f     ⇒ v
                                                   otherwise   ⇒ val
                                      in (p.tag ∈ γ'(val'), γ')
                                      end
                               end
                             end
```

Figure 43: Matching Semantics of the `dflow` Pointcut

4.6.3 Example

To illustrate the semantics of the `dflow` pointcut, let us consider the following example:

Expression:

let $userId = 1$ **in**

let $getInput = \lambda x.\ e_1$ **in** $\quad$ # *getInput* : *gets a user input*

let $write = \lambda x'.\ e_2$ **in** $\quad$ # *write* : *writes a string on a web page*

$z = getInput(userId)$;

$w = write(z)$

The presented example is vulnerable to Cross-Site Scripting (XSS) attacks [29] since an untrusted input received from a user has not been sanitized before being placed into the contents of a web page. Therefore, it enables an attacker to inject malicious scripts into a web page and reveal confidential information. The `dflow` pointcut can be remarkably used to address XSS flaws as shown in [52]. Below, we provide a sanitizing aspect to fix the discussed vulnerability.

Aspect (Pointcuts and Advice):

$$
\begin{array}{lcl}
\mathsf{CallPC}\ p_1 & = & \{id = getInput;\ arg = x\} \\
\mathsf{DFlowPC}\ p_2 & = & \{pc = p_1;\ tag = t\} \\
\mathsf{CallPC}\ p_3 & = & \{id = write;\ arg = y\} \\
\mathsf{AndPC}\ p & = & \{pc_1 = p_2;\ pc_2 = p_3\} \\
\mathsf{Advice}\ a & = & \{body = \ \textbf{let}\ sanitize = \lambda r.\ e_3\ \textbf{in}\ \texttt{proceed}\ (sanitize(y)); \\
 & & \quad pc = p\}
\end{array}
$$

The pointcut p_1 is a `call` pointcut that captures all calls to the *getInput* function.

Likewise, the pointcut p_3 captures all calls to the *write* function. The pointcut p_2 is a `dflow` pointcut that captures all join points that depend on the join points captured by pointcut p_1. Finally, pointcut p picks out all calls to the *write* function that are dependent on the results of invoking the *getInput* function. Advice a, first sanitizes the arguments of join points captured by p and then invokes original join points with the sanitized arguments. More precisely, advice a picks out all calls to $write(z)$ that depend on the result of *getInput* and replaces them with $write(sanitize(z))$ by the following justification:

- The call to $getInput(userId)$ matches p_2 pointcut and consequently the tag t is added to the tagging environment of the function and is given to the result of the function evaluation.
- Then, according to the tag propagation rule for assignment expressions, the value of z gets the tag t.
- Subsequently, the call to $write(z)$ matches the pointcut p since it matches both sub-pointcuts of p. More precisely, it matches the pointcut p_3 as it is a call to the *write* function. It also matches p_2 pointcut as the value of the argument z has the tag t.

Therefore, advice a will be woven at this point and the function *write* will be called with the sanitized input, which is the result of calling $sanitize(z)$.

4.7 Related Work

There are many research contributions that have addressed AOP semantics [3, 4, 10, 13, 17, 23, 25, 30, 41, 51, 53, 85, 86]. Among these contributions, we explore those that are more relevant to our work, mainly contributions that are based on CPS and contributions that target flow-based pointcuts.

Dutchyn [23] has presented a formal model of dynamic join points, pointcuts, and advice using a first-order procedural language called PROC [23]. The proposed semantic model is based on defunctionalization and continuation-passing style. The author has demonstrated that modeling AOP concepts in this style provides a natural way of describing these mechanisms. The proposed model supports `get`, `set`, `call`, and `exec` pointcuts. The author has also provided some hints for implementing the `cflow` pointcut but did not provide the matching algorithm. Compared to [23], our contribution provides a clear presentation allowing a better view of this style of semantics. In addition, we extend the aspect layer with flow-based pointcuts.

Masuhara *et al.* [51] have proposed the point-in-time join point model, where they redefine join points as the moments at the beginning and the end of certain events. Based on this new model, the authors have designed a small AOP language and defined its formal semantics in CPS. Moreover, they demonstrate that this approach is useful to model advanced pointcuts, such as exception handling and control flow. The idea of this work is similar to ours in using continuations to model matching and weaving semantics. However, the main difference is that our semantics is based on frames while in [51] the semantics follows the style of Danvy and Filinski [18] that

represent continuations as λ-functions.

Wand *et al.* [86] have proposed semantics for AOP that handles dynamic join points and recursive procedures. They have provided a denotational semantics for a mini-language that embodies the key features of dynamic join points, pointcuts, and advice. Three kinds of join points were supported, namely `pcall`, `pexecution`, and `aexecution`. The proposed model is implemented as part of Aspect Sandbox (ASB) [24], which is a framework for modeling AOP systems. This model is based on a direct denotational semantics. Consequently, separate data-structures are required for maintaining the dynamic join points while in our semantics the join points arise within the continuation list.

By adopting operational semantics and partial evaluation approaches, Masuhara *et al.* [53] have provided a compilation framework for a simple AOP language named AJD. They have also provided two methods for implementing the `cflow` pointcut, namely, Stack-based and State-based implementations. However, no formal semantics is given for the defined pointcut.

Djoko *et al.* [21] have defined an operational semantics for the main features of AspectJ including `cflow`. The semantics of the `cflow` pointcut presented in this approach is slightly different from AspectJ as they restricted the sub-pointcut to just `call` pointcut. Comparing to this approach, our semantics of the `cflow` pointcut is more general as we support all kinds of pointcuts as a sub-pointcut. In addition, this approach requires additional structures to maintain the join points, which is not the case in our framework.

The `dflow` pointcut was initially proposed by Masuhara and Kawauchi [52]. The

authors have argued about the usefulness of this pointcut in the field of security through an example of a Web-based application. They have also provided the design of the `dflow` pointcut and its matching rules based on the origins of values. The `dflow` pointcut has been implemented as an extension to Aspect Sandbox (ASB) [24]. However, no formal semantics has been provided for this pointcut.

Alhadidi *et al.* [4] have presented the first formal framework for the `dflow` pointcut based on λ-calculus. In this work, dataflow tags are propagated statically to track data dependencies between λ-expressions. Compared to our framework, [4] makes use of the effect-based type system for propagating dataflow tags, matching pointcuts, and weaving advice. Though a static approach can help in reducing the runtime overhead, expressions in this approach need to be typed since matching depends primarily on types. The authors have also provided dynamic semantics and proved that it is consistent with the static semantics. The pointcut enclosed in a `dflow` pointcut is restricted to `call` and `get` pointcuts while we consider the general case in our framework.

4.8 Summary

In this chapter, we have provided formal semantics for aspect matching and weaving. We chose CPS as the basis of our semantics because it provides a concise, accurate, and elegant description of AOP mechanisms. In addition, we have extended our semantic framework with flow-based pointcuts, namely, cflow and dflow pointcuts, since they are important from a security perspective and are widely used to detect

vulnerabilities related to information flow. Using this style of semantics, one can easily notice that CPS and defunctionalization make join points explicit and facilitate the aspect matching and weaving mechanisms.

Chapter 5

Dynamic Aspect Semantics for Executable UML Models

In previous chapter, we presented a formal semantics for aspects matching and weaving for a core language based on λ-calculus. Now, we are ready to apply the same technique on xUML models and present a formal semantics of aspect matching and weaving on executable UML activity diagrams. The target language is Action Language for Foundational UML (Alf) [66] proposed by OMG. In addition to being a standard, Alf is highly expressive and provides precise semantics for specifying detailed behaviors at the modeling level. However, for the sake of illustration, we choose a small core syntax that captures the essence of Alf language as we believe that readability should prevail over completeness.

Similar to the previous chapter, our semantics has a frame-based CPS style as it provides a concise, accurate, and elegant description for modeling aspect-oriented constructs. In our approach, we transfer both the executable activity diagram and

action language expressions into a frame-based representation and provide matching and weaving semantics on frames. In fact, providing a frame-based representation for both UML elements and action language expressions, simplifies and unifies the matching and weaving semantics.

The structure of this chapter is as follows. Section 5.1 presents our proposed syntax and denotational semantics. We transform the semantics into CPS in Section 5.2. In Section 5.3, we extend the language by considering aspect-oriented constructs and subsequently we explore semantics of matching and weaving in Subsection 5.3.2 and Subsection 5.3.3 respectively. In Section 5.4, we extend the semantics with the dataflow pointcut and provide an illustrating example. We discuss related work in Section 5.5. Finally, a summary together with concluding remarks are presented in Section 5.6.

5.1 Syntax and Denotational Semantics

In this section, we present the syntax of UML activity diagrams and Alf language. The notations that are used bellow are introduced in Subsection 2.5. An activity diagram consists of a set of nodes connected by edges. A node can be either an executable node (e.g., action) or a control node (e.g., initial or final). As we mentioned, for the sake of illustration, we choose a small subset of nodes that captures the essence of activity diagrams and omit complex features, such as concurrency and exception handling. Our proposed syntax is shown in Figure 44. The purpose of using labels is to uniquely refer to already defined nodes.

ad	::=	$\bullet \rightarrow n$	**activity**
n	::=	a	**action**
	|	l : `decision` (e, n_1, n_2)	**decision**
	|	l : `merge` $\rightarrow n$	**merge**
	|	$l : \odot$	**activity final**
	|	$a \rightarrow n$	**node sequence**
	|	l	**label**
a	::=	l : `opaque` (e)	**opaque action**
	|	l : `callOp` (f)	**call operation**
	|	l : `read` (x)	**read variable**
	|	l : `write` (x)	**write variable**

Figure 44: Syntax of Activity Diagrams

In the following, we explain the constructs of the syntax:

1. $\bullet \rightarrow n$ denotes an activity diagram where $\bullet$ is the initial node and n is the subsequent flow of nodes.

2. a is an action node, that can be either:

   - l : `opaque` (e), a labeled opaque action where e is an Alf expression specifying its behavior.
   - l : `callOp` (f), a labeled call operation action that invokes an operation f.
   - l : `read` (x), a labeled read variable action that reads the value of x.
   - l : `write` (x), a labeled write variable action that updates the value of x.

3. l : `decision` (e, n_1, n_2) denotes a labeled decision node having two alternative flows n_1 and n_2.

4. l : `merge` $\rightarrow n$ denotes a labeled merge node with the subsequent flow of nodes n.

5. $l : \odot$ denotes a labeled activity final node.

6. $a \rightarrow n$ denotes an action that is connected to the subsequent flow of nodes n.

7. l denotes a label that uniquely refers to a node.

Figure 45 presents the syntax of Alf language. To keep the presentation simple and readable, we choose the main constructs of Alf and remove the object-oriented characteristic of the language.

e	::=	c	**constant**
	|	x	**variable**
	|	$f\,(x) = e$	**operation def.**
	|	$f\,(e)$	**operation call**
	|	`if` e_1 `then` e_2 `else` e_3	**conditional exp.**
	|	e_1; e_2	**sequential exp.**
	|	`new` e	**referencing**
	|	`!` e	**dereferencing**
	|	x `:=` e	**assignment**

Figure 45: Syntax of Alf Language

We consider the following expressions:

- Constants and variables
- Functional constructs
- Conditional expressions
- Sequential expressions
- Imperative features (referencing, dereferencing, and assignment expressions). The expression `new` e allocates a new reference and initializes it with the value of e. The expression !e reads the value stored at the location referenced by the value of e.

The denotational semantics of activity diagrams is presented in Figure 46. The functions and the types are defined in Figure 47.

$$\mathcal{A}[\![\ \bullet\ \rightarrow\ n\ ]\!]\varepsilon\ \sigma = \textbf{let}\ t\ =\ \textit{createToken()}\ \textbf{in}\ \eta[\![\ n\ ]\!]\varepsilon\ \sigma\ t\ ()\ \textbf{end}$$

$$\eta[\![\ l : \texttt{opaque}\ (e)\ ]\!]\varepsilon\ \sigma\ t\ v = \xi[\![\ e\ ]\!]\ \varepsilon\ \sigma$$

$$\eta[\![\ l : \texttt{callOp}\ (f)\ ]\!]\varepsilon\ \sigma\ t\ v = \textbf{let}\ (\langle x, e, \varepsilon'\rangle, \sigma')\ =\ \xi[\![\ \varepsilon(f)\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ \xi[\![\ e\ ]\!]\varepsilon' \dagger [x \mapsto v]\ \sigma'\ \textbf{end}$$

$$\eta[\![\ l : \texttt{read}\ (x)\ ]\!]\varepsilon\ \sigma\ t\ v = \textbf{let}\ (\ell, \sigma') = \xi[\![\ x\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ \ (\sigma'(\ell), \sigma')\ \textbf{end}$$

$$\eta[\![\ l : \texttt{write}\ (x)\ ]\!]\varepsilon\ \sigma\ t\ v = \textbf{let}\ (\ell, \sigma') = \xi[\![\ x\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ \ ((), \sigma' \dagger [\ell \mapsto v])\ \textbf{end}$$

$$\eta[\![\ l : \texttt{decision}\ (e, n_1, n_2)\ ]\!]\varepsilon\ \sigma\ t\ v = \textbf{let}\ (v', \sigma') = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ \textbf{if}\ (v')\ \textbf{then}\ \eta[\![\ n_1\ ]\!]\varepsilon\ \sigma'\ t\ v\ \textbf{else}\ \eta[\![\ n_2\ ]\!]\varepsilon\ \sigma'\ t\ v\ \textbf{end}$$

$$\eta[\![\ l : \texttt{merge} \rightarrow n\ ]\!]\varepsilon\ \sigma\ t\ v = \eta[\![\ n\ ]\!]\varepsilon\ \sigma\ t\ v$$

$$\eta[\![\ l : \odot\ ]\!]\varepsilon\ \sigma\ t\ v = \textbf{let}\ b\ =\ \textit{destroyAllTokens()}\ \textbf{in}\ (v, \sigma)\ \textbf{end}$$

$$\eta[\![\ a \rightarrow n\ ]\!]\varepsilon\ \sigma\ t\ v = \textbf{let}\ (v', \sigma')\ =\ \eta[\![\ a\ ]\!]\varepsilon\ \sigma\ t\ v\ \textbf{in}\ \eta[\![\ n\ ]\!]\varepsilon\ \sigma'\ t\ v'\ \textbf{end}$$

$$\eta[\![\ l\ ]\!]\varepsilon\ \sigma\ t\ v = \eta[\![\ \varepsilon(l)\ ]\!]\varepsilon\ \sigma\ t\ v$$

Figure 46: Denotational Semantics of Activity Diagrams

The semantics that is presented in Figure 46, depicts the behavior of an activity diagram during its execution. Given an activity diagram ad, a dynamic environment ε, and a store σ, the function $\mathcal{A}[\![\ \_\ ]\!]$ yields the computed value v and the updated store σ' after the termination of the activity execution.

When an activity activated, a control token is created by the function *createToken* and placed on the initial node. This token then propagates along the edges to the subsequent nodes. A node starts executing when it gets the required control tokens and data values. Thus, the evaluation function for nodes $\eta[\![\ \_\ ]\!]$ takes a token t and a

$\mathcal{A}[\![\ \_\ ]\!]\ \_\ \_$	:	Activity $\rightarrow$ Env $\rightarrow$ Store $\rightarrow$ Result
$\eta[\![\ \_\ ]\!]\ \_\ \_\ \_\ \_$	:	Node $\rightarrow$ Env $\rightarrow$ Store $\rightarrow$ Token $\rightarrow$ Value $\rightarrow$ Result
$\xi[\![\ \_\ ]\!]\ \_\ \_$	:	Exp $\rightarrow$ Env $\rightarrow$ Store $\rightarrow$ Result
Result	:	Value $\times$ Store
Env	:	Identifier $\rightarrow$ Value
Store	:	Location $\rightarrow$ Value
Value	:	Boolean | Natural | String | Unit | Location | Closure

Figure 47: Semantic Functions and Types

value v as inputs in addition to the environment ε and the store σ.

When the execution of a node terminates, it returns a value, which will be passed to the subsequent nodes through the activity edges, and the updated store. The semantics of an opaque action l : `opaque` (e) depends on the semantics of its Alf expression e. A call operation action l : `callOp` (f) invokes the function f with the argument value v that it receives from its input. A read variable action l : `read` (x) reads the value of the variable x from the store. A write variable action l : `write` (x) updates the value of the variable x with the value v it receives from its input. A decision node l : `decision` $(e,\ n_1,\ n_2)$ guides the flow depending on the value of the condition e. If e evaluates to true, the node n_1 is executed, otherwise the node n_2 is executed. A merge node l : `merge` $\rightarrow n$ passes the token and the data that it receives to its subsequent node n. An activity final node $l : \odot$ terminates the activity execution. Accordingly, all tokens in the activity are destroyed by the function *destroyAllTokens*. Finally, the semantics of a label l depends on the semantics of the referenced node.

$$\xi[\![\ c\ ]\!]\varepsilon\ \sigma = (c, \sigma)$$

$$\xi[\![\ x\ ]\!]\varepsilon\ \sigma = (\varepsilon(x), \sigma)$$

$$\xi[\![\ f\,(x) = e\ ]\!]\varepsilon\ \sigma = (\langle x, e, \varepsilon'\rangle, \sigma)$$

$$\begin{array}{l}\xi[\![\ f\,(e)\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (v, \sigma') = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ \textbf{in} \\ \qquad\qquad \textbf{let}\ (\langle x, e', \varepsilon'\rangle, \sigma'') \ =\ \xi[\![\ \varepsilon(f)\ ]\!]\varepsilon\ \sigma'\ \textbf{in}\ \xi[\![\ e'\ ]\!]\varepsilon' \dagger [x \mapsto v]\ \sigma''\ \textbf{end} \\ \qquad\qquad \textbf{end}\end{array}$$

$$\begin{array}{l}\xi[\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (v, \sigma') = \xi[\![\ e_1\ ]\!]\varepsilon\ \sigma\ \textbf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ (v)\ \textbf{then}\ \xi[\![\ e_2\ ]\!]\varepsilon\ \sigma' \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \xi[\![\ e_3\ ]\!]\varepsilon\ \sigma' \\ \qquad\qquad\qquad\qquad\qquad \textbf{end}\end{array}$$

$$\xi[\![\ e_1;\ e_2\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (v, \sigma') = \xi[\![\ e_1\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ \xi[\![\ e_2\ ]\!]\varepsilon\ \sigma'\ \textbf{end}$$

$$\begin{array}{l}\xi[\![\ \texttt{new}\ e\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (v, \sigma') = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ \textbf{in} \\ \qquad\qquad \textbf{let}\ \ell = \mathit{alloc}(\sigma')\ \textbf{in}\ (\ell, \sigma' \dagger [\ell \mapsto v])\ \textbf{end} \\ \qquad\qquad \textbf{end}\end{array}$$

$$\xi[\![\ !\ e\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (\ell, \sigma') = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ \textbf{in}\ (\sigma'(\ell), \sigma')\ \ \textbf{end}$$

$$\begin{array}{l}\xi[\![\ x\ \texttt{:=}\ e\ ]\!]\varepsilon\ \sigma = \textbf{let}\ (v, \sigma') = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ \textbf{in} \\ \qquad\qquad\qquad \textbf{let}\ (\ell, \sigma'') = \xi[\![\ x\ ]\!]\varepsilon\ \sigma'\ \textbf{in}\ ((), \sigma'' \dagger [\ell \mapsto v])\ \textbf{end} \\ \qquad\qquad\qquad \textbf{end}\end{array}$$

Figure 48: Denotational Semantics of Alf Language

The denotational semantics of Alf language is presented in Figure 48. Given an expression e, a dynamic environment ε, and a store σ, the dynamic evaluation function $\xi[\![\ \_\ ]\!]$ yields the computed value v and the updated store σ'. Notice that in the case of a function definition $f\,(x) = e$, the computed value is a closure $\langle x, e, \varepsilon'\rangle$ capturing the function parameter x, the function body e, and the evaluation environment ε', which maps each free variable of e to its value at the time of the function declaration. The function *alloc* used in the semantics allocates a new cell in the store and returns a reference to it.

5.2 CPS Semantics

In this section, we transform the previously defined denotational semantics into a CPS. As we mentioned earlier, frame-based semantics allows describing matching and weaving processes in activity diagrams and Alf language in a precise and unified way. To help understanding this transformation, we proceed in two steps. First, we elaborate a CPS semantics by representing continuations as functions. Then, we provide CPS semantics by representing continuations as frames. Briefly, continuations describe the semantics of the rest of a computation. Instead of returning a value as in the familiar direct style, a function in CPS style takes another function as an additional argument to which it will pass the current computational result. This additional argument is called a continuation. Continuations are represented as functions, however, for the purpose of modeling join points, we need to move to a frame-based representation. Hence, we perform CPS transformation in two steps.

5.2.1 Representation of Continuations as Functions

We translate the denotational semantics into CPS following the original formulation of the CPS transformation [27]. The continuation κ, represented as a λ-expression, receives the result of the current evaluation and provides the semantics of the rest of the computation. In essence, we modify the evaluation functions to take a continuation as an additional argument. The redefined functions and the types are presented in Figure 49. The CPS semantics of activity diagrams and Alf are presented in Figure 50 and Figure 51 respectively.

$$
\begin{array}{lcl}
\mathcal{A}[\![\ \_\ ]\!]\_\ \_\ \_ & : & \mathsf{Activity} \to \mathsf{Env} \to \mathsf{Store} \to \mathsf{Cont} \to \mathsf{Result} \\
\eta[\![\ \_\ ]\!]\_\ \_\ \_\ \_\ \_ & : & \mathsf{Node} \to \mathsf{Env} \to \mathsf{Store} \to \mathsf{Token} \to \mathsf{Value} \to \mathsf{Cont} \to \mathsf{Result} \\
\xi[\![\ \_\ ]\!]\_\ \_\ \_ & : & \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Store} \to \mathsf{Cont} \to \mathsf{Result} \\
\mathsf{Cont} & : & \mathsf{Result} \to \mathsf{Result}
\end{array}
$$

Figure 49: Redefined Semantic Functions and Types

$$
\begin{array}{l}
\mathcal{A}[\![\ \bullet \ \to \ n\ ]\!]\varepsilon\ \sigma\ \kappa = \textbf{let}\ t\ =\ \mathit{createToken()}\ \textbf{in}\ \eta[\![\ n\ ]\!]\varepsilon\ \sigma\ t\ ()\ \kappa\ \ \textbf{end} \\
\\
\eta[\![\ l : \texttt{opaque}\ (e)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \xi[\![\ e\ ]\!]\ \varepsilon\ \sigma\ \kappa \\
\\
\eta[\![\ l : \texttt{callOp}\ (f)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \xi[\![\ \varepsilon(f)\ ]\!]\varepsilon\ \sigma\ (\lambda(v', \sigma').\ \xi[\![\ e\ ]\!]\varepsilon' \dagger [x \mapsto v]\ \sigma'\ \kappa) \\
\text{where } v' = \langle x, e, \varepsilon' \rangle \\
\\
\eta[\![\ l : \texttt{read}\ (x)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \xi[\![\ x\ ]\!]\varepsilon\ \sigma\ (\lambda(\ell, \sigma').\ \kappa(\sigma'(\ell), \sigma')) \\
\\
\eta[\![\ l : \texttt{write}\ (x)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \xi[\![\ x\ ]\!]\varepsilon\ \sigma\ (\lambda(\ell, \sigma').\ \kappa((), \sigma' \dagger [\ell \mapsto v])) \\
\\
\eta[\![\ l : \texttt{decision}\ (e,\ n_1,\ n_2)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ (\lambda(v', \sigma'). \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ (v')\ \textbf{then}\ \eta[\![\ n_1\ ]\!]\varepsilon\ \sigma'\ t\ v\ \kappa \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \eta[\![\ n_2\ ]\!]\varepsilon\ \sigma'\ t\ v\ \kappa) \\
\\
\eta[\![\ l : \texttt{merge} \to n\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \eta[\![\ n\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa \\
\\
\eta[\![\ l : \odot\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \textbf{let}\ b\ =\ \mathit{destroyAllTokens()}\ \textbf{in}\ \kappa(v, \sigma)\ \ \textbf{end} \\
\\
\eta[\![\ a \to n\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \eta[\![\ a\ ]\!]\varepsilon\ \sigma\ t\ v\ (\lambda(v', \sigma').\ \eta[\![\ n\ ]\!]\varepsilon\ \sigma'\ t\ v'\kappa) \\
\\
\eta[\![\ l\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \eta[\![\ \varepsilon(l)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa
\end{array}
$$

Figure 50: CPS Semantics of Activity Diagrams (Continuations as Functions)

5.2.2 Representation of Continuations as Frames

Continuations, which are λ-expressions, are often represented as closures. Ager *et al.* [2] have provided a systematic conversion of these closures into data structures (or frames) and an apply function interpreting the operations of those closures. This conversion is based on the concept of defunctionalization [73]. The latter is a technique by which higher-order programs, i.e., programs where functions can represent values, are transformed into first-order programs. Each frame stores the value(s) of

$$\xi[\![\ c\ ]\!]\varepsilon\ \sigma\ \kappa = \kappa(c, \sigma)$$

$$\xi[\![\ x\ ]\!]\varepsilon\ \sigma\ \kappa = \kappa(\varepsilon(x), \sigma)$$

$$\xi[\![\ f\,(x) = e\ ]\!]\varepsilon\ \sigma\ \kappa = \kappa(\lambda(v, \kappa').\ [\![\ e\ ]\!]\varepsilon \dagger [x \mapsto v]\ \sigma\ \kappa')$$

$$\xi[\![\ f\,(e)\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ (\lambda(v, \sigma').\ \xi[\![\ \varepsilon(f)\ ]\!]\varepsilon\ \sigma'\ (\lambda(v', \sigma'').\ \xi[\![\ e'\ ]\!]\varepsilon' \dagger [x \mapsto v]\ \sigma''\kappa))$$
where $v' = \langle x, e', \varepsilon' \rangle$

$$\xi[\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e_1\ ]\!]\varepsilon\ \sigma\ (\lambda(v, \sigma').\ \textbf{if}\ (v)\ \textbf{then}\ \xi[\![\ e_2\ ]\!]\varepsilon\ \sigma'\ \kappa\ \textbf{else}\ \xi[\![\ e_3\ ]\!]\varepsilon\ \sigma'\ \kappa)$$

$$\xi[\![\ e_1;\ e_2\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e_1\ ]\!]\varepsilon\ \sigma\ (\lambda(v, \sigma').\ \xi[\![\ e_2\ ]\!]\varepsilon\ \sigma'\ \kappa)$$

$$\xi[\![\ \texttt{new}\ e\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ (\lambda(v, \sigma').\ \textbf{let}\ \ell = \mathit{alloc}(\sigma')\ \textbf{in}\ \kappa(\ell, \sigma' \dagger [\ell \mapsto v]))\ \textbf{end}$$

$$\xi[\![\ !\ e\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ (\lambda(\ell, \sigma').\ \kappa(\sigma'(\ell), \sigma'))$$

$$\xi[\![\ x\ \texttt{:=}\ e\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ (\lambda(v, \sigma').\ \xi[\![\ x\ ]\!]\varepsilon\ \sigma'\ (\lambda(\ell, \sigma'').\ \kappa((), \sigma'' \dagger [\ell \mapsto v])))$$

Figure 51: CPS Semantics of Alf Language (Continuations as Functions)

the free variable(s) of the original continuation function and awaits the value(s) of the previous computation. Following this technique, we transform the continuation functions obtained from the previous step into frames as shown in Figure 52. In the following, we provide details about each frame:

- GetF does not store any value. It awaits a location and a store.
- SetF stores a value. It awaits a location and a store.
- CallF stores a function identifier and an environment. It awaits the value of the function argument.
- ExecF stores the value of the argument. It awaits a closure, which is the result of the evaluation of the function definition, and a store.
- IfF stores then and else expressions and an environment. It awaits the value of

```
type GetF = {}

type SetF = {val : Value}

type CallF = {fun : Identifier;  env : Env}

type ExecF = {arg : Value}

type IfF = {thenExp : Exp;  elseExp : Exp;  env : Env}

type DecisionF = {thenNode : Node;  elseNode : Node;
                  env : Env;  token : Token;  val : Value}

type ExpSeqF = {nextExp : Exp;  env : Env}

type NodeSeqF = {nextNode : Node;  env : Env; token : Token}

type AllocF = {}

type RhsF = {id : Identifier;  env : Env}
```

Figure 52: Frames

the condition and a store.

- DecisionF stores then and else nodes, an environment, a control token, and a value. It awaits the value of the condition and a store.

- ExpSeqF stores the next expression and an environment. It awaits the value of the first expression and a store.

- NodeSeqF stores the next node, an environment, and a control token. It awaits the output value of the first node and a store.

- AllocF does not store any value. It awaits the value to be stored in the newly allocated cell and a store.

- RhsF stores an identifier and an environment. It awaits a location and a store.

Using frame-based semantics, the continuation κ consists of a list of frames. Before presenting the semantics, we first define the primitive functions that will be used. The primitive *push* extends a continuation list with another frame. (Figure 53)

$$
\begin{array}{l}
push \quad : \quad \mathsf{Frame} \rightarrow \mathsf{Cont} \rightarrow \mathsf{Cont} \\
\\
\textbf{let } push\ f\ \kappa = f :: \kappa
\end{array}
$$

Figure 53: Apply Function

The primitive *apply*, defined in Figure 54, pops the top frame from a continuation list and evaluates it based on its corresponding continuation function. When the list becomes empty, the primitive *apply* returns the current value and the store as a result.

$$
\begin{array}{lll}
apply & : & \mathsf{Cont} \rightarrow (\mathsf{Value} \times \mathsf{Store}) \rightarrow (\mathsf{Value} \times \mathsf{Store}) \\
\textbf{let } apply\ \kappa\ (v, \sigma) = \textbf{match } \kappa \textbf{ with} & & \\
\qquad [\,] & \Rightarrow & (v, \sigma) \\
\quad \mid f :: \kappa' & \Rightarrow & \mathcal{F}[\![\, f \,]\!]\sigma\ v\ \kappa'
\end{array}
$$

Figure 54: Apply Function

The frame-based semantics of the activity diagrams is presented in Figure 55 and the frame-based semantics of Alf is presented in Figure 56. Figure 57 shows the evaluation of the frames that are needed for computations.

$$\mathcal{A}[\![\ \bullet\ \rightarrow\ n\ ]\!]\varepsilon\ \sigma\ \kappa = \textbf{let}\ t\ =\ \mathit{createToken()}\ \textbf{in}\ \eta[\![\ n\ ]\!]\varepsilon\ \sigma\ t\ ()\ \kappa\ \textbf{end}$$
$$\eta[\![\ l : \texttt{opaque}\ (e)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \xi[\![\ e\ ]\!]\ \varepsilon\ \sigma\ \kappa$$

$$\eta[\![\ l : \texttt{callOp}\ (f)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \mathit{apply}(\mathit{push}(\mathsf{CallF}(f, \varepsilon), \kappa), (v, \sigma))$$

$$\eta[\![\ l : \texttt{read}\ (x)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \xi[\![\ x\ ]\!]\varepsilon\ \sigma\ (\mathit{push}(\mathsf{GetF}(), \kappa))$$

$$\eta[\![\ l : \texttt{write}\ (x)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \xi[\![\ x\ ]\!]\varepsilon\ \sigma\ (\mathit{push}(\mathsf{SetF}(v), \kappa))$$

$$\eta[\![\ l : \texttt{decision}\ (e, n_1, n_2)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ (\mathit{push}(\mathsf{DecisionF}\ (n_1, n_2, \varepsilon, t, v), \kappa))$$

$$\eta[\![\ l : \texttt{merge} \rightarrow n\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \eta[\![\ n\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa$$

$$\eta[\![\ l : \odot\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \textbf{let}\ b\ =\ \mathit{destroyAllTokens()}\ \textbf{in}\ \kappa(v, \sigma)\ \textbf{end}$$

$$\eta[\![\ a \rightarrow n\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \eta[\![\ a\ ]\!]\varepsilon\ \sigma\ t\ v\ (\mathit{push}(\mathsf{NodeSeqF}(n, \varepsilon, t), \kappa))$$
$$\eta[\![\ l\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa = \eta[\![\ \varepsilon(l)\ ]\!]\varepsilon\ \sigma\ t\ v\ \kappa$$

Figure 55: Frame-Based Semantics of Activity Diagrams

$$\xi[\![\ c\ ]\!]\varepsilon\ \sigma\ \kappa = \mathit{apply}(\kappa, (c, \sigma))$$

$$\xi[\![\ x\ ]\!]\varepsilon\ \sigma\ \kappa = \mathit{apply}(\kappa, (\varepsilon(x), \sigma))$$

$$\xi[\![\ f\,(x) = e\ ]\!]\varepsilon\ \sigma\ \kappa = \mathit{apply}(\kappa, (\langle x, e, \varepsilon' \rangle, \sigma))$$

$$\xi[\![\ f\,(e)\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ (\mathit{push}(\mathsf{CallF}(f, \varepsilon),\ \kappa))$$

$$\xi[\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e_1\ ]\!]\varepsilon\ \sigma\ (\mathit{push}(\mathsf{IfF}(e_2, e_3, \varepsilon), \kappa))$$

$$\xi[\![\ e_1;\ e_2\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e_1\ ]\!]\varepsilon\ \sigma\ (\mathit{push}(\mathsf{ExpSeqF}(e_2, \varepsilon), \kappa))$$

$$\xi[\![\ \texttt{new}\ e\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ (\mathit{push}(\mathsf{AllocF}(), \kappa))$$

$$\xi[\![\ !\ e\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ (\mathit{push}(\mathsf{GetF}(), \kappa))$$

$$\xi[\![\ x\ \texttt{:=}\ e\ ]\!]\varepsilon\ \sigma\ \kappa = \xi[\![\ e\ ]\!]\varepsilon\ \sigma\ (\mathit{push}(\mathsf{RhsF}(x, \varepsilon), \kappa))$$

Figure 56: Frame-Based Semantics of Alf Language

$$\mathcal{F}[\![\ \mathsf{GetF}\ f\ ]\!]\sigma\ v\ \kappa = \mathit{apply}(\kappa, (\sigma(v), \sigma))$$

$$\mathcal{F}[\![\ \mathsf{SetF}\ f\ ]\!]\sigma\ v\ \kappa = \mathit{apply}(\kappa, ((), \sigma \dagger [v\ \mapsto f.\mathit{val}]))$$

$$\mathcal{F}[\![\ \mathsf{CallF}\ f\ ]\!]\sigma\ v\ \kappa = \xi[\![\ (f.\mathit{env})(f.\mathit{fun})\ ]\!](f.\mathit{env})\ \sigma\ (\mathit{push}(\mathsf{ExecF}(v), \kappa))$$

$$\mathcal{F}[\![\ \mathsf{ExecF}\ f\ ]\!]\sigma\ v\ \kappa = \xi[\![\ e\ ]\!]\varepsilon' \dagger [x\ \mapsto f.\mathit{arg}]\ \sigma\ \kappa \quad \text{where } v = \langle x, e, \varepsilon' \rangle$$

$$\begin{aligned}\mathcal{F}[\![\ \mathsf{IfF}\ f\ ]\!]\sigma\ v\ \kappa = {} & \textbf{if}\ (v)\ \textbf{then}\ \xi[\![\ f.\mathit{thenExp}\ ]\!](f.\mathit{env})\ \sigma\ \kappa \\ & \textbf{else}\ \xi[\![\ f.\mathit{elseExp}\ ]\!](f.\mathit{env})\ \sigma\ \kappa\end{aligned}$$

$$\begin{aligned}\mathcal{F}[\![\ \mathsf{DecisionF}\ f\ ]\!]\sigma\ v\ \kappa = {} & \textbf{if}\ (v)\ \textbf{then}\ \eta[\![\ f.\mathit{thenNode}\ ]\!](f.\mathit{env})\ \sigma\ (f.\mathit{token})\ (f.\mathit{val})\ \kappa \\ & \textbf{else}\ \eta[\![\ f.\mathit{elseNode}\ ]\!](f.\mathit{env})\ \sigma\ (f.\mathit{token})\ (f.\mathit{val})\ \kappa\end{aligned}$$

$$\mathcal{F}[\![\ \mathsf{ExpSeqF}\ f\ ]\!]\sigma\ v\ \kappa = \xi[\![\ f.\mathit{nextExp}\ ]\!](f.\mathit{env})\ \sigma\ \kappa$$

$$\mathcal{F}[\![\ \mathsf{NodeSeqF}\ f\ ]\!]\sigma\ v\ \kappa = \eta[\![\ f.\mathit{nextNode}\ ]\!](f.\mathit{env})\ \sigma\ (f.\mathit{token})\ v\ \kappa$$

$$\mathcal{F}[\![\ \mathsf{AllocF}\ f\ ]\!]\sigma\ v\ \kappa = \textbf{let}\ \ \ell = \mathit{alloc}(\sigma)\ \ \textbf{in}\ \mathit{apply}(\kappa, (\ell, \sigma \dagger [\ell \mapsto v]))\ \textbf{end}$$

$$\mathcal{F}[\![\ \mathsf{RhsF}\ f\ ]\!]\sigma\ v\ \kappa = \xi[\![\ f.\mathit{id}\ ]\!](f.\mathit{env})\ \sigma\ (\mathit{push}(\mathsf{SetF}(v), \kappa))$$

Figure 57: Semantics of Frames

5.3 Aspect Syntax and Semantics

In this section, we present our aspect language and elaborate its semantics. We start by presenting the aspect syntax. Then, we elaborate the matching and the weaving semantics.

5.3.1 Aspect Syntax

An aspect, depicted in Figure 58, includes a list of advice. Advice specifies actions to be performed when join points satisfying a particular pointcut are reached. In our approach, join points are specific points in the execution of both activity and Alf expressions. Syntactically, advice contains two parts: (1) a body, which is an

expression and (2) a pointcut, which designates a set of join points. Advice can be applied before, after, or around a join point. However, before and after advice can be expressed as around advice using the `proceed` expression. Hence, we consider all kinds of advice as around advice as this does not restrict the generality of the approach. A pointcut designates a set of join points. We first consider basic pointcuts: `GetPC`, `SetPC`, `CallPC`, and `ExecPC`. The pointcut `GetPC` (resp. `SetPC`) picks out join points where the value of a variable is got from (resp. set to) the store. The pointcut `CallPC` (resp. `ExecPC`) picks out join points where a function is called (resp. executed).

type Aspect	=	Advice `list`
type Advice	=	$\{body$: Exp; pc : Pointcut$\}$
type Pointcut	=	GetPC | SetPC | CallPC | ExecPC | NotPC — AndPC
type GetPC	=	$\{id$: Identifier$\}$
type SetPC	=	$\{id$: Identifier; val : Value$\}$
type CallPC	=	$\{id$: Identifier; arg : Identifier$\}$
type ExecPC	=	$\{id$: Identifier; arg : Identifier$\}$
type NotPC	=	$\{pc$: Pointcut$\}$
type AndPC	=	$\{pc_1$: Pointcut; pc_2 : Pointcut$\}$

Figure 58: Aspect Syntax

As in AspectJ [45], advice may also compute the original join point through a special expression named `proceed`. Hence, as shown in Figure59, we extend the core syntax with an additional expression `proceed` (e) to denote the computation of the original join point with possibly a new argument e.

e	::=	...	
	|	`proceed` (e)	**proceed**

Figure 59: The `proceed` Expression

5.3.2 Matching Semantics

Matching is a mechanism for identifying the join points that are targeted by an advice. In our approach, join points correspond to specific points in the execution of activity diagrams actions and Alf expressions. However, since the execution semantics is presented in a frame-based style, both kinds of join points are continuation frames and arise naturally within the semantics. Therefore, our matching semantics, as shown in Figure 60, examines whether a continuation frame satisfies a given pointcut or not.

$$
\begin{array}{l}
\mathit{match\_pc} : \mathsf{Pointcut} \rightarrow \mathsf{Frame} \rightarrow \mathsf{Value} \rightarrow \mathsf{Store} \rightarrow \mathsf{Env} \rightarrow \mathsf{Cont} \rightarrow \mathsf{Boolean} \\
\\
\textbf{let}\ \mathit{match\_pc}\ p\ f\ v\ \sigma\ \varepsilon\ \kappa = \textbf{match}\ (p, f)\ \textbf{with} \\
\quad (\mathsf{GetPC}\ p, \mathsf{GetF}\ f) \quad \Rightarrow \quad \varepsilon(p.id) = v \\
\quad \mid (\mathsf{SetPC}\ p, \mathsf{SetF}\ f) \quad \Rightarrow \quad \varepsilon(p.id) = v \\
\quad \mid (\mathsf{CallPC}\ p, \mathsf{CallF}\ f) \quad \Rightarrow \quad p.id = f.\mathit{fun} \\
\quad \mid (\mathsf{ExecPC}\ p, \mathsf{ExecF}\ f) \quad \Rightarrow \quad \textbf{let}\ (v', \sigma') = \xi[\![\ \varepsilon(p.id)\ ]\!]\ \varepsilon\ \sigma\ \kappa\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ v = v' \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{end} \\
\quad \mid (\mathsf{NotPC}\ p, \mathsf{Frame}\ f) \quad \Rightarrow \quad \textbf{not}\ \mathit{match\_pc}(p.pc, f, v, \sigma, \varepsilon, \kappa) \\
\quad \mid (\mathsf{AndPC}\ p, \mathsf{Frame}\ f) \quad \Rightarrow \quad \mathit{match\_pc}(p.pc_1, f, v, \sigma, \varepsilon, \kappa)\ \textbf{and} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathit{match\_pc}(p.pc_2, f, v, \sigma, \varepsilon, \kappa) \\
\quad \mid \textbf{otherwise} \quad \Rightarrow \quad \textbf{false}
\end{array}
$$

Figure 60: Matching Semantics

Given a pointcut p, the current frame f, the current value v, a store σ, an environment ε, and a continuation κ, the matching semantics examines whether f matches p. Matching depends on three factors, the kind and the content of the frame f and the current value v that f receives. In the case of:

- GetPC, there is a match if f is a GetF frame and the location of the identifier given in p is equal to the location that f receives.
- SetPC, there is a match if f is a SetF frame and the location of the identifier given in p is equal to the location that f receives.
- CallPC, there is a match if f is a CallF frame and it holds a function equal to the one given in p.
- ExecPC, there is a match if f is an ExecF frame and the evaluation of the function given in p is equal to the closure that f receives.
- NotPC, there is a match if f does not match the sub-pointcut of p. (The sub-pointcut of pointcut p is the pointcut, which is enclosed in p)
- AndPC, there is a match if f matches both its sub-pointcuts.

5.3.3 Weaving Semantics

The weaving semantics describes how to apply matching the advice at the identified join points. Since join points correspond to frames, the advice body provides a means to modify the behavior of those frames. The weaving is performed automatically during the execution. Therefore, we redefine the apply function, as shown in Figure

61, to take an aspect α and an environment ε into account. Similarly, the signatures of the functions are also modified.

```
apply : Cont → (Value × Store) → Env → Aspect → (Value × Store)

let apply κ (v, σ) ε α = match κ with

        [ ]   ⇒   (v, σ)
    | f :: κ′  ⇒   let ms = get_matches(f, v, σ, ε, α, κ′) in
                    if ms = [ ] then F⟦ f ⟧ε σ v α κ′
                    else
                      let argV = match f with
                          SetF f        ⇒ f.val
                        | CallF f       ⇒ v
                        | ExecF f       ⇒ f.arg
                        | otherwise     ⇒ ()
                      in execute_advice(ms, f, argV, σ, ε, α, κ′)
                      end
                end
```

Figure 61: Redefined Apply Function

The weaving is done in two steps. When a frame is activated, we first check for a matching advice by calling the *get_matches* function. If there is any applicable advice, the function *execute_advice* is called. Otherwise, the original computation is performed. In the following, we explain these two steps.

Advice Matching

Advice matching is shown in Figure 62.

To get applicable advice, we go through the aspect and check whether their enclosed pointcuts match the current frame. This is done by calling the function *match_pc* defined previously in Figure 60. In case there is a match, we return a structure MatchedAD containing the advice itself and the pointcut arguments that will pass values to the advice.

```
type MatchedAD   =  {arg : Identifier; ad : Advice}
get_matches      :  Frame → Value → Store → Env → Aspect → Cont
                    → MatchedAD list

let get_matches f v σ ε α κ = match α with

      [ ]   ⇒ [ ]
|ad :: α'   ⇒ let p = ad.pc in
                if match_pc(p, f, v, σ, ε, α, κ) then
                  let arg = match p with
                      SetPC p              ⇒ p.id
                    | CallPC p | ExecPC p ⇒ p.arg
                    | otherwise           ⇒ () in
                    MatchedAD(arg, ad) :: get_matches(f, v, σ, ε, α', κ)
                  end
                else get_matches(f, v, σ, ε, α', κ)
              end
```

Figure 62: Advice Matching

Advice Execution

Advice execution is shown in Figure 63. It starts by evaluating the first applicable advice. The remaining pieces of advice as well as the current frame are stored in the environment by binding them to auxiliary variables &*proceed* and &*jp* respectively. To evaluate the advice body, we define a new frame, AdvExecF, as follows:

type AdvExecF = {*matches* : MatchedAD `list`; *jp* : Frame}

$\mathcal{F}[\![$ AdvExecF f $]\!]\varepsilon\ \sigma\ v\ \alpha\ \kappa = execute\_advice(f.matches, f.jp, v, \sigma, \varepsilon, \alpha, \kappa)$

The evaluation of `proceed` is provided below. The value of its argument is passed to the next advice or to the current join point if there is no further advice. To execute the remaining advice, the AdvExecF frame is added to the frame list.

$[\![$ `proceed` (e) $]\!]\varepsilon\ \sigma\ \alpha\ \kappa = [\![\ e\ ]\!]\varepsilon\ \sigma\ \alpha\ (push(\text{AdvExecF}(\varepsilon(\&proceed), \varepsilon(\&jp)), \kappa))$

$$
\begin{array}{l}
\mathit{execute\_advice} : \mathsf{MatchedAD\ list} \to \mathsf{Frame} \to \mathsf{Value} \to \mathsf{Store} \to \mathsf{Env} \\
\qquad\qquad \to \mathsf{Aspect} \to \mathsf{Cont} \to \mathsf{Result} \\
\\
\textbf{let}\ \mathit{execute\_advice}\ ms\ f\ v\ \sigma\ \varepsilon\ \alpha\ \kappa = \textbf{match}\ \ ms\ \textbf{with} \\
\qquad [\,] \quad \Rightarrow \quad \mathit{apply}(\mathit{push}(\mathsf{MarkerF}(), (\mathit{push}(f, \kappa))), (v, \sigma), \varepsilon, \alpha) \\
\\
\quad |m :: ms' \quad \Rightarrow \quad \textbf{let}\ ad = m.ad\ \textbf{in} \\
\qquad\qquad\qquad\quad \xi[\![\ ad.body\ ]\!]\varepsilon \dagger [\&\mathit{proceed} \mapsto ms', \\
\qquad\qquad\qquad\quad \&\mathit{jp} \mapsto f, m.arg \mapsto v]\ \sigma\ \alpha\ \kappa \\
\qquad\qquad\qquad\quad \textbf{end}
\end{array}
$$

Figure 63: Advice Execution

When all applicable pieces of advice are executed, the original computation, i.e., the current frame is invoked. To avoid matching the currently matched frame repeatedly, we introduce a new frame, MarkerF, which invokes the primary apply function, renamed here as *apply_prim*.

$$
\begin{array}{l}
\textbf{type}\ \mathsf{MarkerF} = \{\ \} \\
\\
\mathcal{F}[\![\ \mathsf{MarkerF}\ f\ ]\!]\varepsilon\ \sigma\ v\ \alpha\ \kappa = \ \mathit{apply\_prim}(\kappa, (v, \sigma))
\end{array}
$$

5.4 Semantics of the Dataflow Pointcut

In this section, we extend our framework with the semantics of the dataflow pointcut (`dflow`) [52]. This pointcut is useful from a security perspective since it can detect important vulnerabilities that are related to information flow, such as Cross-site Scripting (XSS) [15] and SQL injection [83].

The `dflow` pointcut picks out join points based on the origins of values, i.e., `dflow`$[x,\ x']$$(p)$ matches a join point if the value of x originates from the value of x'. Variable x should be bound to a value in the current join point whereas variable

x' should be bound to a value in a past join point matched by p.

To match `dflow` pointcuts, particular tags are assigned to the `dflow` pointcuts to discriminate `dflow` pointcuts and track dependencies between values [52]. Briefly, if an expression matches the sub-pointcut of a `dflow` pointcut, p, this expression is tagged with the tag of this `dflow` pointcut. This tag is then propagated to other expressions that are data-dependent on the expression that matches the sub-pointcut. As defined below, the `dflow` pointcut has a sub-pointcut pc and a unique tag that discriminates it from other `dflow` pointcuts.

type DFlowPC = $\{pc$: Pointcut; tag : Identifier$\}$

In order to track dependencies between values, we use a tagging environment γ that maps values to tags. Tag propagation is performed dynamically during the execution of the activity diagram and Alf expressions. In particular, this is done at the frames side as shown in Figure 64. Notice that now the functions take the tagging environment γ as an additional argument, however their definitions remain the same. Notice also that in the case of an ExecF frame, the closure $v = \langle x, e, \varepsilon', \gamma' \rangle$ is extended with a tagging environment γ' to capture the tags generated during the function execution. In addition, we define a marker frame DflowF that is used for tag propagation in the case of a function call. The DflowF frame stores a tagging environment before entering a function call and awaits the result of the call.

type DflowF = $\{tag\_env$: Env$\}$

In the following, we explain the tag propagation rules for the affected frames:

$$\mathcal{F}[\![\ \mathsf{GetF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathit{apply}(\kappa, (\sigma(v), \sigma), \varepsilon, \gamma \dagger [\sigma(v) \mapsto \gamma(v)], \alpha)$$

$$\mathcal{F}[\![\ \mathsf{SetF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathit{apply}(\kappa, ((), \sigma \dagger [v\ \mapsto f.\mathit{val}]), \varepsilon, \gamma \dagger [v\ \mapsto \gamma(f.\mathit{val})], \alpha)$$

$$\mathcal{F}[\![\ \mathsf{CallF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \xi[\![\ (f.\mathit{env})(f.\mathit{fun})\ ]\!](f.\mathit{env})\ \gamma\ \sigma\ \alpha\ (\mathit{push}(\mathsf{ExecF}(v), \kappa))$$

$$\begin{aligned}\mathcal{F}[\![\ \mathsf{ExecF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \xi[\![\ e\ ]\!](\varepsilon' \dagger [x\ \mapsto f.\mathit{arg}])\\ (\gamma' \dagger [\varepsilon(x) \mapsto \gamma(f.\mathit{arg})])\ \sigma\ \alpha\ (\mathit{push}(\mathsf{DflowF}(\gamma), \kappa))\end{aligned}$$
where $v = \langle x, e, \varepsilon', \gamma' \rangle$

$$\begin{aligned}\mathcal{F}[\![\ \mathsf{IfF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathbf{if}\ (v)\ \mathbf{then}\ \xi[\![\ f.\mathit{thenExp}\ ]\!](f.\mathit{env})\ \gamma\ \sigma\ \alpha\ \kappa\\ \mathbf{else}\ \xi[\![\ f.\mathit{elseExp}\ ]\!](f.\mathit{env})\ \gamma\ \sigma\ \alpha\ \kappa\end{aligned}$$

$$\begin{aligned}\mathcal{F}[\![\ \mathsf{DecisionF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathbf{if}\ (v)\ \mathbf{then}\ \eta[\![\ f.\mathit{thenNode}\ ]\!](f.\mathit{env})\ \gamma\ \sigma\ (f.\mathit{token})\ (f.\mathit{val})\ \alpha\ \kappa\\ \mathbf{else}\ \eta[\![\ f.\mathit{elseNode}\ ]\!](f.\mathit{env})\ \gamma\ \sigma\ (f.\mathit{token})\ (f.\mathit{val})\ \alpha\ \kappa\end{aligned}$$

$$\mathcal{F}[\![\ \mathsf{ExpSeqF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \xi[\![\ f.\mathit{nextExp}\ ]\!](f.\mathit{env})\ \gamma\ \sigma\ \alpha\ \kappa$$

$$\mathcal{F}[\![\ \mathsf{NodeSeqF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \eta[\![\ f.\mathit{nextNode}\ ]\!](f.\mathit{env})\ \gamma\ \sigma\ (f.\mathit{token})\ v\ \alpha\ \kappa$$

$$\begin{aligned}\mathcal{F}[\![\ \mathsf{AllocF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathbf{let}\ \ \ell = \mathit{alloc}(\sigma)\ \ \mathbf{in}\\ \mathit{apply}(\kappa, (\ell, \sigma \dagger [\ell \mapsto v]), \varepsilon, \gamma \dagger [\ell \mapsto \gamma(v)], \alpha)\\ \mathbf{end}\end{aligned}$$

$$\mathcal{F}[\![\ \mathsf{RhsF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \xi[\![\ f.\mathit{id}\ ]\!](f.\mathit{env})\ \gamma\ \sigma\ \alpha\ (\mathit{push}(\mathsf{SetF}(v), \kappa))$$

$$\mathcal{F}[\![\ \mathsf{AdvExecF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathit{execute\_advice}(f.\mathit{matches}, f.\mathit{jp}, v, \sigma, \varepsilon, \gamma, \alpha, \kappa)$$

$$\mathcal{F}[\![\ \mathsf{MarkerF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathit{apply\_prim}(\kappa, (v, \sigma))$$

$$\mathcal{F}[\![\ \mathsf{DFlowF}\ f\ ]\!]\varepsilon\ \gamma\ \sigma\ v\ \alpha\ \kappa = \mathit{apply}(\kappa, (v, \sigma), \varepsilon, f.\mathit{tag\_env} \dagger [v \mapsto \mathit{getTags}(\gamma)], \alpha)$$

Figure 64: Semantics of Frames with the `dflow` Pointcut

- In the case of a GetF frame, the tags of the location v propagate to the value stored at that location.

- In the case of a SetF frame, the tags of the value of the right-hand side of an assignment stored in the frame propagate to the location of the assignment identifier.

- In the case of a ExecF frame, the tags of the argument value $f.arg$ propagate to the value of the variable x. In addition, the tags of the argument and the

tags that are generated during the function execution propagate to the result of the function. For this reason, we use a DflowF frame to access the result of the function call and restore the tagging environment after returning from the call. The function $getTags(\gamma)$ used in $\mathcal{F}[\![\ \mathsf{DFlowF}\ f\ ]\!]$ retrieves all the tags stored in the tagging environment γ.

- In the case of an AllocF frame, the tags of the value v propagate to the created location ℓ.

The matching semantics of the `dflow` pointcut is presented in Figure 65. A join point frame f matches a `dflow` pointcut that contains a pointcut pc and a tag t if: (1) the frame f matches the pointcut pc of the `dflow` pointcut, or (2) the set of tags of the value that the frame f awaits (captured by the variable val') contains the tag t. In case a frame f matches the pointcut pc of the `dflow` pointcut, the tag t propagates to the value associated with the frame f (captured by the variable val).

5.4.1 Example

To illustrate the `dflow` pointcut, let us consider the *SearchPage* activity diagram presented in Figure 66. It starts by accepting a search request. Then, the searched phrase is extracted by the *GetQuery* operation. If the requested phrase is empty, an error message is generated. Otherwise, the *Search* action is executed and the result message, containing both the requested phrase and the search result, is generated. Finally, the generated message is printed on the web page.

The presented example is vulnerable to XSS attacks since the untrusted input received from the user has not been sanitized before being placed into the contents

```
type JpF = GetF | SetF | CallF | ExecF
let match_pc p f v σ ε γ α κ = match (p, f) with
...
| (DFlowPC p, JpF f) ⇒
  let (b, γ') = match_pc(p.pc, f, v, σ, ε, γ, α, κ) in
    let val = match f with
      GetF f    ⇒ v
      SetF f    ⇒ f.val
      CallF f   ⇒ let (v', σ') = ξ⟦ ε(f.fun) ⟧ε γ σ α κ in
                    v'
                  end
      ExecF f   ⇒ v
    in
      if (b)
      then (true, γ' † [val ↦ γ'(val) ∪ {p.tag}])
      else let val' = match f with
                CallF f    ⇒ v
                otherwise  ⇒ val
           in (p.tag ∈ γ'(val'), γ')
           end
    end
  end
```

Figure 65: Matching Semantics of the `dflow` Pointcut

of the web page. Therefore, it enables an attacker to inject malicious scripts into the web page and reveal confidential information. To fix this vulnerability, we need to sanitize the untrusted input and all data that originated from it before printing them on the web page. The `dflow` pointcut can be remarkably used to address this problem. As mentioned before, the `dflow` pointcut, `dflow`(p), picks out all points in the activity execution where values are dependent on the join points that are previously picked out by p. Therefore, by defining pointcut p as CallPC(*GetQuery*), `dflow`(p) picks all join points that are originated from the search phrase, which is the user input. Below, we provide a sanitizing aspect for fixing the discussed vulnerability.

Aspect (Pointcuts and Advice):

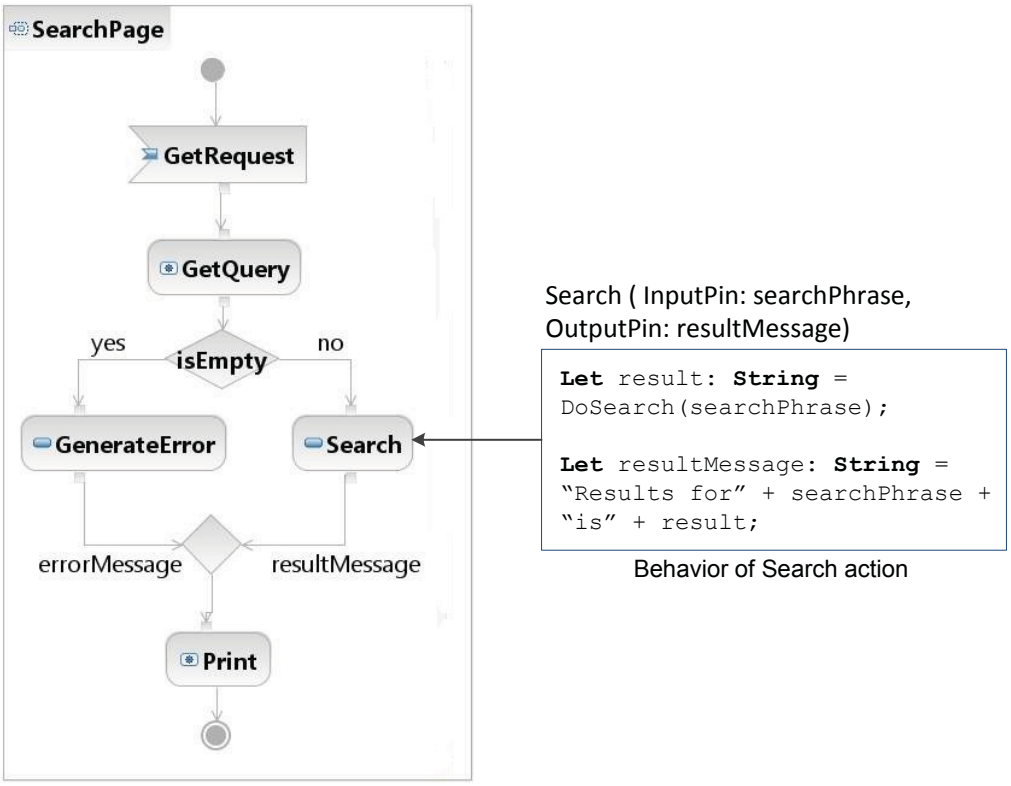

Figure 66: Dflow Example

$$\text{CallPC } p_1 \quad = \quad \{id = GetQuery;\ \ arg = x\}$$

$$\text{DFlowPC } p_2 \quad = \quad \{pc = p_1;\ \ tag = t\}$$

$$\text{CallPC } p_3 \quad = \quad \{id = Print;\ \ arg = y\}$$

$$\text{AndPC } p_4 \quad = \quad \{pc_1 = p_2;\ \ pc_2 = p_3\}$$

$$\text{Advice } a \quad = \quad \{body = \texttt{proceed}\ (Sanitize(y));\ \ pc = p_4\}$$

Briefly, the aspect captures points where the *Print* operation is called with an argument that is originated from the user input. The aspect first sanitizes the argument by calling the *Sanitize* operation and then calls the *Print* operation with the sanitized argument based on the following justification:

- The *GetQuery* operation matches p_2 since it matches its subpointcut p_1. Consequently, the tag t is added to the tagging environment of the function and is given to the result of the function evaluation.

- Then, if the search phrase is not empty, the *Search* action is executed. According to the tag propagation rule for assignment and call operation expressions, the values of the variables *result* and *resultMessage* get the tag t.

- Subsequently, the *Print* operation matches p_4 since it matches both its subpointcuts. More precisely, it matches p_3 as it is a call to the *Print* operation. And also matches p_2 as the value of its argument has the tag t. Therefore, the sanitizing advice will be woven at this point.

5.5 Related Work

We categorize related work into three main areas:

(1) **AOM and xUML:** Fuentes and Sanchez [31] have extended UML to support aspect-oriented concept by proposing Aspect-Oriented Executable Modelling (AOEM) UML 2.0 Profile. In this profile, an aspect is modeled as a UML class which contains common methods and distinct methods as advice. The behaviors of advice pieces are modeled as activity diagrams and injected into the base model as structured activities. Additionally, pointcuts are defined as composition rules specified with sequence diagrams. The pointcuts matching process is based on method interception technique. Briefly, it observes sending and receiving messages and injects corresponding advice when a certain condition is satisfied during execution of a model. Based on

the profile, the authers designed and built an aspect-oriented dynamic model weaver that can be used for running aspect-oriented models where aspects are woven during model execution. This weaver supports adding new behaviors before, after, and around join points, but does not support removing behaviors. In addition, this approach considers a small subset of actions defined in executable UML elements and does not support complicated actions such as Opaque action which allows modelers to specify behaviors using Alf expressions. Also, due to their application of method interception technique, advice can merely be injected before, after or around method calls. Moreover, there are no theoretical foundations for the defined approach.

Zhang et al. [91] have presented Motorola WEAVR; a tool for weaving aspects into executable UML state machines. The aspects are woven into the base models following two ways: (1) wrapping where original join points are replaced by an operation call to the corresponding advices, and (2) inlining where advices are inlined in the base model. This weaver supports two types of join points: actions and transitions. However, this weaver is based on the Telelogic TAU G2 implementation, which makes it tool-dependent and not portable. Additionally, there are also no theoretical foundations for the defined approach.

Jackson *et al.* have introduced an approach for specifying and weaving KerThemes. [40] A KerTheme comprises an executable class diagram and a sequence diagram. This weaver is based on KerMeta action language [60] to define precise behaviors and provide executability. It only supports weaving of executable class diagrams as all behavioral diagrams, such as sequence diagrams, are defined as methods. Furthermore, KerMeta has been designed for specifying meta-model behaviors and it is not

as expressive as UML action languages.

(2) **AOP Semantics:** There are many research contributions that have addressed AOP semantics [3, 4, 10, 17, 23, 25, 30, 51, 53, 85, 86]. Among these contributions, we explore those that are more relevant to our work, mainly contributions that are based on CPS semantics.

Dutchyn [23] has presented a formal model of dynamic join points, pointcuts, and advice using a first-order procedural language called PROC [23]. The proposed semantic model is based on defunctionalization and continuation-passing style. The author has demonstrated that modeling join points, pointcuts, and advices in a frame-based continuation-passing style provides a natural way of describing these mechanisms. The proposed model supports `get`, `set`, `call`, and `exec` pointcuts. The author has also provided some hints for implementing the `CFlow` pointcut but did not provide the matching algorithm. Compared to this work, our contribution provides a clean representation allowing a better view of this style of semantics. In addition, we extend the syntax with imperative features to handle references and assignments. Moreover, in addition to the basic pointcuts, our framework provides also semantics of the `CFlow` and `DFlow` pointcuts.

Masuhara *et al.* [51] have proposed the point-in-time join point model, where they redefine join points as the moments at the beginning and the end of certain events. Based on this new model, the authors have designed a small AOP language and defined its formal semantics in CPS style. Moreover, they demonstrated that this approach is useful to model advanced pointcuts, such as, exception handling and control flow. The idea of this work is similar to ours in using continuations to model

matching and weaving semantics. However, the main difference is that our semantics is based on frames while in [51], the semantics follows the style of Danvy and Filinski [18] that represent continuations as λ-functions.

Wand *et al.* [86] have proposed the first semantics for AOP that handles dynamic join points and recursive procedures. They have provided a denotational semantics for a mini-language that embodies the key features of dynamic join points, pointcuts, and advice. Three kinds of join points were supported, namely `pcall`, `pexecution`, and `aexecution`. The proposed model is implemented as part of Aspect Sandbox (ASB) [24], which is a framework for modeling AOP systems. This model is based on a direct denotational semantics. Consequently, separate data-structures are required for maintaining the dynamic join points, while in our semantics the join points arise within the continuation structure.

(3) **The dflow pointcut:** The `dflow` pointcut was initially proposed by Masuhara [52]. He presented the design of the pointcut and its prototype implementation. Also he argued about the usefulness of the pointcut specially in the field of security. In [4] the authors presented a formal framework for the `dflow` pointcut based on lambda calculus. A static and a dynamic semantics are elaborated for tags propagations and proved to be consistent.

5.6 Summary

In this chapter, we have presented a semantic framework for aspect matching and weaving on executable UML activity diagrams, including dflow pointcut, which is

important from a security perspective. We chose CPS as it provides a concise and elegant description of aspect-oriented mechanisms. In fact, one can easily notice that CPS and defunctionalization make join points explicit and facilitate aspect matching and weaving. In addition, frame-based representation unifies these processes for both activity diagrams elements and Alf expressions.

Chapter 6

Conclusion

In our previous work, briefly presented in Chapter 3, we implemented an Aspect-Oriented Modeling (AOM) framework for weaving crosscutting concerns into UML models. In this research work, we decided to enhance our approach and provide a framework for aspect matching and weaving on Executable UML models (xUML) as such models are expected to play a significant role in the future of software modeling. There are two main motives for taking this decision. First, executable models enable security experts to enrich their security aspect libraries and provide aspects with more precise behaviors. Second, such models allow the security experts to provide more advanced security aspects (such as an aspect for capturing data dependencies) due to their detailed behavior specifications and execution capability.

However, since a xUML model is a combination of UML elements and code written in an action language, neither AOM nor AOP approaches are merely enough for addressing crosscutting concerns in such models. In fact, we need to come up with an approach that handles matching and weaving on both UML elements and code. in

order to break down the problem, we decided to first focus on providing a semantics for aspects matching and weaving for a core language based on λ-calculus (Chapter 4) and then apply the technique on xUML models and deal with both model elements and codes simultaneously (Chapter 5).

Our semantics has a frame-based CPS style as it provides a concise, accurate, and elegant description for modeling aspect-oriented constructs. In our proposed semantics for aspect matching and weaving on executable UML activity, we transferred both the executable activity diagram and action language expressions into a frame-based representation and defined matching and weaving semantics on frames. In fact, providing a frame-based representation for both UML elements and action language expressions simplifies and unifies the matching and weaving semantics. In addition, we have extended our semantic framework with DFlow pointcuts, since it is important from a security perspective and is widely used to detect vulnerabilities related to information flow.

As a future work, we plan to extend our framework by considering other activity elements, such as fork, join, and exception handling. Also, such semantics framework can be further used to prove some key properties or to establish some internal consistency properties. It is worth noting that, we leave the implementation of the proposed semantic framework as a future work since Alf language has not been finalized. In addition, there were no execution engines available for Alf at the time we conducted this work.

Bibliography

[1] OCaml for Scientists. Available at `http://caml.inria.fr/pub/docs/manual-ocaml`, 2011.

[2] M. S. Ager, O. Danvy, and J. Midtgaard. A Functional Correspondence Between Monadic Evaluators and Abstract Machines for Languages with Computational Effects. *Theoretical Computer Science*, 342:04–28, 2005.

[3] D. Alhadidi, N. Belblidia, M. Debbabi, and P. Bhattacharya. An AOP Extended Lambda-Calculus. In *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM'2007)*, pages 183–194. IEEE Computer Society, 2007.

[4] D. Alhadidi, A. Boukhtouta, N. Belblidia, M. Debbabi, and P. Bhattacharya. The Dataflow Pointcut: A Formal and Practical Framework. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, (AOSD'09), pages 15–26, New York, NY, USA, 2009. ACM.

[5] J. H. Allen. *Software Security Engineering: A Guide for Project Managers*. Sei Series in Software Engineering. Addison-Wesley, 2008.

[6] A. W. Appel. *Compiling with Continuations (corr. version)*. Cambridge University Press, 2006.

[7] Aspect-Oriented Modeling Workshop. http://www.aspect-modeling.org/. Last visited: January 2010.

[8] B. Blakley, C. Heath, and members of The Open Group Security Forum. Security Design Patterns. Technical Report G031, Open Group, 2004.

[9] R. Bodkin. Enterprise Security Aspects. In *Proc. of the 4th Workshop on AOSD Technology for Application-Level Security*, 2004.

[10] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. ABC: A Minimal Aspect Calculus. In *Proceedings of the International Conference on Concurrency Theory*, volume 3170 of *LNCS*, pages 209–224. Springer, 2004.

[11] R. C. Seacord D. Svoboda K. Togashi C. Dougherty, K. Sayre. Secure Design Patterns. Technical Report, CMU/SEI-2009-TR-010, ESC-TR-2009-010, Software Engineering Institute, Carnegie Mellon University, 2009.

[12] M-T. Chan and L-F. Kwok. Integrating Security Design into the Software Development Process for E-commerce Systems. *Information Management & Computer Security*, 9(3):112–122, 2001.

[13] C. Clifton and G. T. Leavens. MiniMAO: An imperative core language for studying aspect-oriented reasoning. *Sci. Comput. Program.*, 63(3):321–374, 2006.

[14] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of Foundations of Software Engineering*, pages 88–98. ACM Press, 2001.

[15] Cross-site Scripting (XSS). https://www.owasp.org/index.php/Cross-site_Scripting_XSS. Last visited: June 2012.

[16] L. Dai and K. Cooper. Modeling and Analysis of Non-Functional Requirements as Aspects in a UML Based Architecture Design. In *Proceedings of the 6th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks*, pages 178–183, Washington, DC, USA, 2005. IEEE Computer Society.

[17] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A Polymorphic Aspect-Oriented Functional Programming Language. *ACM Transactions on Programming Languages and Systems*, 30:14:1–14:60, 2008.

[18] O. Danvy and A. Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP'90, pages 151–160, New York, NY, USA, 1990. ACM.

[19] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *PPDP*, pages 162–174. ACM, 2001.

[20] T. Dierks and E. Rescorla. The transport layer security (tls) protocol. In *IETF RFC 4346*, 2006.

[21] S. D. Djoko, R. Douence, P. Fradet, and D. Le Botlan. CASB: Common Aspect Semantics Base - AOSD Europe Deliverable No. 41, 2006.

[22] T. Doan, L. D. Michel, and S. A. Demurjian. A Formal Framework for Secure Design and Constraint Checking in UML. In *Proceedings of the International Symposium on Secure Software Engineering (ISSSE'06)*, Washington, DC, 2006.

[23] C. Dutchyn. Specializing Continuations: a Model for Dynamic Join Points. In *Proceedings of the 6th International Workshop on Foundations of Aspect-Oriented Languages*, pages 45–57. ACM, 2007.

[24] C. Dutchyn, G. Kiczales, and H. Masuhara. Aspect SandBox. Available at `http://www.cs.ubc.ca/labs/spl/projects/asb.html`, 2002.

[25] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and Scoping of Aspects in Higher-Order Languages. *Science of Computer Programming*, 63:207–239, 2006.

[26] E. B. Fernandez and R. Warrier. Remote Authenticator/Authorizer. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP'03)*, 2003.

[27] M. J. Fischer. Lambda Calculus Schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109, New York,USA, 1972. ACM.

[28] Fortify. Available at https://www.fortify.com/downloads2/user/Fortify_Case_For_Application_Security.pdf. Last visited: June 2012.

[29] Fortify. Software security, protect your software at the source. Available at: `http://www.fortify.com/vulncat/en/vulncat/IPV.html`, 2011.

[30] B. De Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: Flexible and Safe Pointcut/Advice Bindings. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, AOSD'08, pages 60–71, New York, NY, USA, 2008. ACM.

[31] L. Fuentes and P. Sánchez. Transactions on aspect-oriented software development vi. chapter Dynamic Weaving of Aspect-Oriented Executable UML Models, pages 1–38. Springer-Verlag, Berlin, Heidelberg, 2009.

[32] S. Gao, Y. Deng, H. Yu, X. He, K. Beznosov, and K. Cooper. Applying Aspect-Orientation in Designing Security Systems: A Case Study. In *Proceedings of International Conference of Software Engineering and Knowledge Engineering*, 2004.

[33] G. Georg, R. B. France, and I. Ray. An Aspect-Based Approach to Modeling Security Concerns. In J. Jürjens, M. V. Cengarle, E. B. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 Workshop*, pages 107–120. Technische Universität München, Institut für Informatik, 2002.

[34] G. Georg, S. H. Houmb, and I. Ray. Aspect-Oriented Risk-Driven Development of Secure Applications. In Ernesto Damiani and Peng Liu, editors, *Proceedings of the 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec'2006)*, volume 4127 of *Lecture Notes in Computer Science*, pages 282–296. Springer, 2006.

[35] G. Georg, I. Ray, K. Anastasakis, B. Bordbar, M. Toahchoodee, and S. H.

Houmb. An aspect-oriented methodology for designing secure applications. *Information & Software Technology*, 51(5):846–864, 2009.

[36] M. J. C. Gordon. *Programming language theory and its implementation - applicative and imperative paradigms.* Prentice Hall International series in Computer Science. Prentice Hall, 1988.

[37] Mentor Graphics. Object Action Language Reference Manual. Available at http://www.mentor.com/products/sm/techpubs/object-action-language -reference-manual-38098, 2009.

[38] K. S. Hoo, A. W. Sudbury, and A. R. Jaquith. Tangible ROI through Secure Software Engineering. *Secure Business Quarterly*, 1(2), Fourth Quarter 2001.

[39] J. Vijayan. Available at:http://www.computerworld.com/s/article/ 9136805/SQL_injection_attacks_led_to_Heartland_Hannaford_breaches. Last visited: June 2012.

[40] A. Jackson, J. Kleinand B. Baudry, and S. Clarke. KerTheme: Testing Aspect Oriented Models. In *Proceedings of the ECMDA Wshop. on Integration of Model Driven Development and Model Driven Testing.*, 2006.

[41] R. Jagadeesan, A. Jeffrey, and J. Riely. A Calculus of Untyped Aspect-Oriented Programs. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 54–73. Springer-Verlag, 2003.

[42] L. Brown Jr, F. L. Brown, J. Divietri, G. Diaz De Villegas, and E. B. Fernandez. The Authenticator Pattern. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP'03)*, page 6. Wiley, 1999.

[43] J. Jürjens and S. H. Houmb. Dynamic Secure Aspect Modeling with UML: From Models to Code. In L. C. Briand and C. Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 142–155. Springer, 2005.

[44] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327–353, London, UK, 2001. Springer-Verlag.

[45] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. pages 327–353. Springer-Verlag, 2001.

[46] G. Kiczales, J. Lamping, A. Mendhekar, Ch. Maeda, C. V. Lopes, J-M Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[47] P. J. Landin. A Generalization of Jumps and Labels. In *Report, UNIVAC Systems Programming Research*, 1965.

[48] C. L. Lazar, I. Lazar, B. Parv, S. Motogna, and I. G. Czibula. Using a fuml action language to construct uml models. In *Proceedings of the 2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC '09, pages 93–101, Washington, DC, USA, 2009. IEEE Computer Society.

[49] Kennedy Carter Limited. UML ASL Reference Guide. Available at`http://www.ooatool.com/docs/ASL03.pdf`, 2003.

[50] T. Lodderstedt, D. Basin, and J. Doser. Model-Driven Security: from UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):39–91, 2006.

[51] H. Masuhara, Y. Endoh, and A. Yonezawa. A Fine-Grained Join Point Model for More Reusable Aspects. In *Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS'2006)*, volume 4279 of *LNCS*, pages 131–147, 2006.

[52] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In *Proceedings of the first Asian Symposium on Programming Languages and Systems (APLAS)*, pages 105–121, 2003.

[53] H. Masuhara, G. Kiczales, and Ch. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 46–60, Berlin, Heidelberg, 2003. Springer-Verlag.

[54] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[55] M. S. Merkow and L. Raghavan. *Secure and Resilient Software Development*. Auerbach Publications, London, UK, 2010.

[56] W. De Meuter and N. Boyen. An Informal Tour On Denotational Semantics. Technical Report vub-prog-tr-94-08, Programming Technology Lab, Vrije Universiteit Brussel, 1994.

[57] C. Montangero, M. Buchholtz, L. Perrone, and S. Semprini. For-LySa: UML for Authentication Analysis. In *Global Computing: IST/FET International Workshop (GC'04)*, volume 3267 of *Lecture Notes in Computer Science*, pages 93–106. Springer Verlag, 2005.

[58] D. Mouheb, Ch. Talhi, M. Nouh, V. Lima, M. Debbabi, L. Wang, and M. Pourzandi. Aspect-oriented modeling for representing and integrating security concerns in uml. In *Proceedings of the International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 197–213, 2010.

[59] A. Mourad, M. A. Laverdière, and M. Debbabi. A High-Level Aspect-Oriented Based Framework for Software Security Hardening. *Information Security Journal: A Global Perspective*, 17(2):56–74, 2008.

[60] P. A. Muller, F. Fleurey, and J. M. Jzquel. Weaving Executability into OO Metalanguages. In *Intl. Conference on Model Driven Engineering Languages and Systems, LNCS 3713*, pages 264–278. Springer, 2005.

[61] M. Nouh, R. Ziarati, D. Mouheb, D. Alhadidi, M. Debbabi, L. Wang, and M. Pourzandi. Aspect Weaver: A Model Transformation Approach for UML Models. In *Proceedings of the 2010 conference of the Centre for Advanced Studies on Collaborative Research*, pages 139–153, 2010.

[62] Object Management Group (OMG). Meta Object Facility Specification, Version 2.0, 2006.

[63] Object Management Group (OMG). Object Constraint Language Specification, Version 2.0, 2006.

[64] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0, 2008.

[65] Object Management Group (OMG). Unified Modeling Language (OMG UML): Superstructure, Version 2.4.1, 2011.

[66] Object Management Group (OMG). Action Language for Foundational UML (ALF): Concrete Syntax for UML Action Language. Available at`http://www.omg.org/spec/ALF/`, 2011.

[67] Object Management Group (OMG). Semantics Of A Foundational Subset For Executable UML Models (FUML). Available at`http://www.omg.org/spec/FUML/`, 2011.

[68] OWASP Top 10. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. Last visited: September 2012.

[69] J. Pavlich-Mariscal, T. Doan, L. Michel, S. Demurjian, and T. Ting. Role Slices: A Notation for RBAC Permission Assignment and Enforcement. In *Proceedings of the 19th Annual IFIP WG 11.3*, pages 40–53, Connecticut, USA, 2005.

[70] J. Pavlich-Mariscal, L. Michel, and S. Demurjian. Enhancing UML to Model Custom Security Aspects. In *Proceedings of the 11th International Workshop on Aspect-Oriented Modeling (AOM@AOSD'07)*, 2007.

[71] I. Ray, R. France, N. Li, and G. Georg. An Aspect-Based Approach to Modeling Access Control Concerns. *Information and Software Technology*, 46(9):575–587, 2004.

[72] I. Ray, N. Li, D. K. Kim, and R. France. Using Parameterized UML to Specify and Compose Access Control Models. In *Proceedings of the 6th IFIP TC-11 WG 11.5 Working Confrence on Integrity and Internal Control in Information Systems (IICIS'03)*, Lausanne, Switzerland, 2003.

[73] J. C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*, volume 2 of *ACM'72*, pages 717–740, New York, NY, USA, 1972. ACM.

[74] J. C. Reynolds. The Discoveries of Continuations. *Journal of Lisp and Symbolic Computation, Special issue on continuations*, 6(3-4), 1993.

[75] S. Romanosky. Enterprise Security Design Patterns. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP'02)*, 2002.

[76] D. A. Schmidt. *Denotational semantics: a methodology for language development.* William C. Brown Publishers, Dubuque, IA, USA, 1986.

[77] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering (Wiley Software Patterns Series)*. John Wiley & Sons, 2006.

[78] Pathfinder Solutions. Platform Independent Action Language. Available at`http://www.ooatool.com/docs/PAL04.pdf`, 2004.

[79] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *Proceedings of the 40th International Conference on Tools Pacific (CRPIT'02)*, pages 53–60, Darlinghurst, Australia, 2002.

[80] State of Software Security Report Volume 3. http://www.veracode.com/reports/index.html. Last visited: September 2012.

[81] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1981.

[82] C. Strachey and C. P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. technical monograph prg 11, oxford university computing laboratory, 1974.

[83] The Open Web Application Security Project (OWASP). https://www.owasp.org/index.php/SQL_Injection. Last visited: June 2012.

[84] J. Viega, J. T. Bloch, and P. Chandra. Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*, 14:31–39, 2001.

[85] D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. volume 38 of *ICFP'03*, pages 127–139, New York, NY, USA, 2003. ACM.

[86] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 26:890–910, 2004.

[87] B. De Win. Engineering Application Level Security through Aspect-Oriented Software Development. PhD Thesis, Katholieke Universiteit Leuven, 2004.

[88] M. Woodside, D. C. Petriu, D. B. Petriu, J. Xu, T. Israr, G. Georg, R. France, J. M. Bieman, S. H. Houmb, and J. Jürjens. Performance Analysis of Security Aspects by Weaving Scenarios Extracted from UML Models. *Journal of Systems and Software*, 82(1):56–74, 2009.

[89] S. G. Yi, Y. Deng, H. Yu, X. He, K. Beznosov, and K. Cooper. Applying Aspect-Orientation in Designing Security Systems: A Case Study. In *Proceedings of the International Conference of Software Engineering and Knowledge Engineering*, 2004.

[90] G. Zhang, H. Baumeister, N. Koch, and A. Knapp. Aspect-Oriented Modeling of Access Control in Web Applications. In *Proceedings of the 6th Workshop on Aspect Oriented Modeling*, 2005.

[91] J. Zhang, T. Cottenier, A. Berg, and J. Gray. Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. *Journal of Object Technology. Special Issue on AOM*, 6(7):89–108, 2007.