Aspects with Program Analysis for Security Policies

Yang, Fan; Nielson, Flemming; Nielson, Hanne Riis

Publication date:
2010

Document Version
Publisher's PDF, also known as Version of record

Aspects with Program Analysis for Security Policies

Fan Yang

Kongens Lyngby 2010
IMM-PHD-2010-239

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

Enforcing security policies to IT systems, especially for a mobile distributed system, is challenging. As society becomes more IT-savvy, our expectations about security and privacy evolve. This is usually followed by changes in regulation in the form of standards and legislation. In many cases, small modification of the security requirement might lead to substantial changes in a number of modules within a large mobile distributed system. Indeed, security is a crosscutting concern which can spread to many business modules within a system, and is difficult to be integrated in a modular way.

This dissertation explores the principles of adding challenging security policies to existing systems with great flexibility and modularity. The policies concerned cover both classical access control and explicit information flow policies. We built our solution by combining aspect-oriented programming techniques with static program analysis techniques. The former technique can separate security concerns out of the main logic, and thus improves system modularity. The latter can analyze the system behavior, and thus helps detect software bugs or potential malicious code.

We present AspectKE, an aspect-oriented extension of the process calculus KLAIM that excels at modeling mobile, distributed systems. A novel feature of our approach is that advices are able to analyze the future use of data, which is achieved by using program analysis techniques. We also present AspectK to propose other possible aspect-oriented extensions based on KLAIM, followed by a discussion of open joinpoints that commonly exist in coordination languages such as KLAIM. Based on the idea of AspectKE, we design and implement a proof-of-concept programming language AspectKE*, which enables program-

mers to easily specify analysis-based security policies with the help of high-level program analysis predicates and functions. The prototype is efficiently realized by a two-stage implementation strategy and a static-dynamic dual value evaluation mechanism. We have performed two case studies to evaluate our programming model and language design. One application is based on a electronic health care workflow system. The other is a distributed chat system. We considered a number of security policies for both primary and secondary use of data, classical access control and predictive access control - control access based on the future behavior of a program. Some of the above mentioned policies can only be enforced by analysis of process continuations.

Resumé

Det er en vigtig udfordring at kunne håndhæve sikkerhedspolitikker i mobile og distribuerede IT-systemer. Efterhånden som samfundet bliver mere og mere afhængigt af IT stiger vores forventninger til sikkerhed og privathed, hvilket leder til ny lovgivning og nye standarder. I mange tilfælde giver selv små ændringer i sikkerhedskravene anledning til ganske omfattende ændringer i større distribueret systemer. Det skyldes at sikkerhed går på tværs af de sædvanlige modularitets-principper og derfor ikke kan håndteres modulært.

Denne afhandling udforsker metoder, der muliggør en flexible og modulær tilføjelse af nye sikkerheds-politikker til eksisterende IT-systemer. Det gælder både klassiske politikker for adgangskontrol men også politikker for informations flow. Metoden er baseret på brugen af aspekt-orienteret programmering sammen med teknikker fra statisk analyse af programmer. Her gør den aspekt-orientede tilgang det muligt at adskille sikkerheds-overvejelserne fra de funktionelle overvejelser og dermed opnå øget modularitet. Brugen af statisk analyse gør det muligt at finde en del af de mulige sikkerhedshuller, der overlever denne design-proces.

Afhandlingen præsenterer AspectKE, der er en aspekt-orienteret udvidelse af process algebraen KLAIM, der er specielt god til at modellere mobile og distribuerede IT-systemer. En nyskabelse er muligheden af at aspekter kan analysere den fremtidige brug af data ved hjælp af teknikker fra statisk analyse af programmer. Vi præsenterer også AspectK som et eksempel på andre aspekt-orienterede udvidelser og studerer de såkaldte åbne join-punkter, der er almindelige i KLAIM og andre sprog til koordination. Med udgangspunkt i AspectKE designes og udvikles der et prototype programmeringssprog, AspectKE*, som gør det muligt at programmere sikkerheds-politikker, der udnytter de nye muligheder. Prototypen er implementeret i to trin ved hjælp af en særlig statisk-

dynamisk evaluerings-mekanisme. Vi har afprøvet systemet på to større eksempler. Den ene modellerer et elektronisk patient journal system og den anden er et distribueret chat system. I begge tilfælde ser vi på et antal sikkerhedspolitikker for både primær og sekundær brug af data såvel som klassiske politikker for adgangskontrol og informations flow baseret adgangskontrol, der netop afhænger af den fremtidige brug af data. Evnen til at analysere fremtidige beregninger er essentiel for at kunne udtrykke visse af disse politikker.

Preface

This dissertation was prepared at the department of Informatics and Mathematical Modelling, the Technical University of Denmark, in partial fulfillment of the requirements for acquiring the Ph.D. degree in Computer Science. The Ph.D study has been carried out under the supervision of Professor Flemming Nielson and Professor Hanne Riis Nielson in the period from June 2007 to August 2010. The Study was funded by DTU Scholarship but also got partially supported by Danish Strategic Research Council (project 2106-06-0028) project *Aspects of Security for Citizens*.

Most of the work behind this dissertation has been carried out independently and I take full responsibility for its contents. During the three year study, my excellent research collaborators have also given me many inspirations and suggestions. Besides work with my two supervisors, the development of process calculi AspectKE (Chapters 3 and 4), reported in [YHNN] and AspectK (Chapter 5), reported in [HNNY08], were in collaboration with Chris Hankin; The development of programming language AspectKE* (Chapters 6 7 and 8), reported in [YMA+10a, YMA+10b, YAM+], were in collaboration with Hidehiko Masuhara and Tomoyuki Aotani.

Lyngby, October 2010

Fan Yang

Acknowledgements

I would like to thank my supervisors, Flemming Nielson and Hanne Riis Nielson, for providing me with the opportunity to work on this interesting research project, for their excellent guidance, inspiring suggestions and insightful comments on my work, during my entire three year PhD study.

I would like to thank the rest of the LBT group: Christian Probst, Henrik Pilegaard, Terkel Tolstrup, Christoffer Rosenkilde Nielsen, Jörg Bauer, Sebastian Nanz, Han Gao, Ye Zhang, Ender Yuksel, Nataliya Skrypnyuk, Matthieu Queva, Alejandro Hernandez, Fuyuan Zhan, Piotr Filipiuk, Kebin Zeng, Roberto Larcher, Michael J.A. Smith, Sebastian Mödersheim, Lijun Zhang, Jose Quaresma, and Michal Tomasz Terepeta, for creating an inspiring and friendly working environment. I got many good suggestions from them of research and of life.

I would like to thank Chris Hankin. I really enjoyed the time working with him on process calculi design. I really appreciate the series of fruitful meetings both at the Technical University of Denmark and Imperial College London, which greatly speed up my work. I also would like to thank him for proof-reading the dissertation, and giving many useful comments.

I would like to thank Hidehiko Masuhara, for hosting and supervising me during my five months external visit at the PPP research group in the University of Tokyo, Japan, and for inviting me for a second visit to finish the remaining research work. I really benefitted from those fruitful discussions and constructive comments on my work, especially for unmatched suggestions over programming language design and implementation.

I would like to thank Tomoyuki Aotani, for his supreme guidance on implementing static analysis on bytecode instructions using ASM framework, and the inspiring discussions about analysis-based pointcuts.

I would like to thank the rest of the PPP research group: Robert Hirschfeld, Watanabe Takuya, Muroi Hiroaki, Kouhei Sakurai, Kazunori Kawauchi and Manabu Toyama, for their comments about my work and for help me to adjust to the Japanese style of daily life.

I would like to thank Lorenzo Bettini for developing the excellent Klava system, and discussing it with me.

I would like to thank the members from *Aspects of Security for Citizens* project: Christian Probst, Hubert Baumeister, Sebastian Nanz, Michael Huth, and Alejandro Hernandez for their comments on my work. Special thanks to Alejandro Hernandez, we really had some nice time on proving interesting analysis properties of AspectKE.

I would also like to thank the evaluation committee: Rocco De Nicola, Mario Südholt, and Christian Probst, for their helpful comments on this dissertation.

Last but not the least, I would like to thank my family: my father, my mother, especially my beloved wife Ziyan Feng, for her proof-reading of the dissertation, and for her consistent support and patience during my entire PhD study.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Ensuring Security in Distributed Systems. In modern society, ensuring security of IT systems, especially for a distributed system, is challenging. This is because in such a system, trusted components have to work with untrusted components. For example, while a user of a chat system trusts the programs running at the service provider's computers, he or she might want to run a client program developed by an untrusted third-party. In such a case, we need to ensure that the untrusted third-party program does not perform malicious operations. For another example, in an electronic health record (EHR) workflow system, while we allow users to define and perform workflow processes on patients' medical records, we must ensure the user-defined processes, which can not be fully trusted, do not illegally access and/or use the patients' private sensitive information intentionally or unintentionally. This dissertation will tackle the issue of enforcing challenge security policies to such untrusted, distributed and mobile systems.

Coordination models and languages such as *tuple space* systems [Gel85, FAH99] provide a very elegant and simple way of building distributed systems. The core characteristics of a tuple space system is the shared network-based space (tuple space) that serves as both data (tuple) storage and data exchange area. Its simple yet expressive distributed primitives can easily operate on these data through the shared space. As the tuple space systems have simple yet powerful language features for building distributed and collaborative applications, we choose it to illustrate our security solutions for distributed systems.

Separating Security Concerns from Main Logic. Although there is broad agreement that security and other non-functional properties should be designed and integrated into IT systems from the beginning, it is also recognized that, as society becomes more IT-savvy, our expectations about security and privacy evolve continuously. This is usually followed by changes in regulation in the form of standards and legislation. On the other hand, the functional aspects of IT systems change evolve as well, which requires security and privacy concerns to be able to easily adapted and enforced to evolving systems.

Following the good software engineering practice that one can only obtain adequate security if it is catered in the original design of system rather than being added as an afterthought, we would still agree that security should feature in the initial design of a system. However, we shall also argue that there is merit in separating out security and other non-functional properties so that they can be updated without disturbing the functional aspects of the system. One objective of this dissertation is to explore the principles behind separating security concerns out of the functional aspects of a distributed system with appropriate language support, so that the security concerns could be easily added to existing programs at various development phases, e.g, from the beginning, or after the system has been fully developed and deployed.

The traditional approach to enforcing security policies is to use a *reference monitor* [Gol99] that dynamically tracks the execution of the program. It makes appropriate checks on each basic operation being performed, either blocking the operation or allowing it to proceed. In concrete systems this is implemented as part of the operating system or as part of the interpreter for the language at hand (e.g. the Java byte code interpreter); in both cases as part of the trusted computing base. Sometimes it is found to be more cost effective to systematically modify the code so as to explicitly perform the checks that the reference monitor would have imposed [ES00]. In any case, even small modification in the security policies may involve substantial changes in the code for checks and the underlying system.

The *aspect-oriented programming* [KHH+01,KLM+97] is an interesting approach to separate concerns. This technology enables us to program non-functional concerns in the so-called *aspect* program, which could then be merged into the main program and enforce the properties defined by the aspect program. Security is an obvious candidate for such non-functional concerns, because e.g., the security policy can be implemented by more skilled or more trusted programmers, or indeed because security considerations can be retrofitted by (re)defining the aspect to suit the (new) security policies. The detailed definition of the aspects will then make decisions about how to possibly modify the operation being trapped. This calls for a modified language (like AspectJ [KHH+01] for Java) that supports the use of aspects and incorporates a notion of trapping oper-

ations and applying advice. It is possible to systematically modify the code so as to explicitly perform the operations that the advice would have imposed (e.g. [KHH+01]).

In many cases the aspect-oriented approach provides a more flexible way for dealing with modifications in security policies [Dan07, GDY+04, GRF02, PS08, WJP02] than the use of reference monitors. It facilitates to develop frameworks for enforcing security policies that may be well suited to the task at hand but that are perhaps not of general applicability and therefore not appropriate for incorporating into a reference monitor. In this dissertation, we take the aspect-oriented approach as the basic policy enforcement mechanism, and our focus is to explore and improve the type of security policies that can be naturally expressed in aspects.

Enforcing Explicit Information Flow Policies. There exists a variety of security policies proposed and used in modern IT systems. When inspecting them from the data-usage point of view and taking EHR system as an example, many security policies regulate the *primary use of data* (patient data collected for use in direct health care service), but there is a trend for the *secondary use of data* (re-use existing data for new purposes like public health care research, commercial activities) and thus a number of policies are proposed for regulating their usage [And00, Can02, SBH+07]. The latter is more challenging as we need to explicitly address issues of data access and information (data) flow, possibly throughout the entire data life cycle, which is hard to deal with by traditional access control methods. Therefore the enforcement of policies that rely on explicit information (data) flow is one of the main themes in this dissertation.

One way of enforcing data-flow policies is the *runtime monitoring* approach. Essentially, we keep track of execution state of each variable. Although the data-flow information can be precisely recorded, this approach produces large execution time overhead and lacks mechanisms to look into future events.

On the contrary, the *static analysis* approach is a collection of techniques that can check a program before execution [NNH05]. They could simulate all execution paths of a program, including future events, and could make sufficiently precise safe-approximation of the properties of a program statically. Their innate characteristics suggest they could obtain data-flow information of a program earlier than the runtime monitoring approach, which sometimes is much more preferable. After all, detecting and preventing the execution of malicious/unsafe code in the first place can save potential cost of failure recovery. In this dissertation, we combine static analysis approaches and aspect-oriented programing to enforce challenging data-flow security policies.

1.1 Preliminary Summary

The main thesis of this dissertation is to show that

> Aspect-oriented programming provides a flexible way of enforcing security policies in distributed systems, more specifically, within the tuple space paradigm. Static program analysis techniques can enhance the expressiveness of security aspects and elegantly support the enforcement of security policies that rely on information flow.

For this purpose, we proceed by formally designing and practically implementing an aspect-oriented programming language on top of the tuple space system, where static program analysis components have been elegantly integrated. We considered and applied our language to two application domains to show its usefulness, which are studies about enforcing security policies on relatively a large scale application such as an EHR workflow system, and on a lightweight application such as a distributed chat system.

More concretely, first we design a core process calculus AspectKE based on KLAIM [DFP98], which is equipped with formal syntax and semantics so as to clarify the essential idea of our approach. We choose an EHR workflow application as a running example to illustrate the design features of the AspectKE programming model. Second, we discuss the aspect-oriented tuple space programming model in a more general setting, by presenting the design of another process calculus AspectK and the concept of open joinpoint. Third, we extend AspectKE into a more practical prototype language AspectKE*, developed an efficient runtime system AspectKlava, and demonstrate the usefulness and practicality of the theoretical model developed in the first part. A distributed chat system is mainly used to illustrate the practical work.

Though AspectKE, AspectK and AspectKE* are based on KLAIM, the techniques developed in this dissertation can also be applied to other distributed frameworks, especially those based on process calculi. We believe it is useful for monitoring, analyzing and controlling the behavior of the mobile processes, under a distributed AOP execution environment.

1.2 Dissertation Outline

This dissertation consists of, apart from the introduction, eight chapters.

Chapter 2 presents the general research background behind our language design.

In particular, the coordination language KLAIM [DFP98] is presented and chosen as the basis for our work, because of its favorable features for implementing mobile and distributed systems.

Chapter 3 starts to present the formal syntax and semantics of the process calculus AspectKE, an aspect-oriented extension of KLAIM. This chapter only focuses on the basic features of this language, while more advanced features are presented in Chapter 4. Here we show how aspects can trap the actions of KLAIM, how the language is useful for enforcing a number of classical access control models, and how multiple security policies can be integrated into existing systems thereby allowing policies to be refined at a later stage. The enforcement of primary use of data policies are demonstrated in this chapter.

Chapter 4 continues Chapter 3 by presenting the advanced features of AspectKE. Here we show how aspects can trap processes of KLAIM and exploit the ability to analyze the behavior of remotely executed processes as well as the future use of data in the current process. The enforcement of secondary use of data policies and predictive access control policies are discussed in this chapter.

Chapter 5 presents an extension to the basic part of AspectKE, namely AspectK. This chapter also discusses other possible aspect-oriented extensions of the tuple space system and argues that open joinpoints commonly exist in coordination languages.

Chapter 6 and the following chapters present the practical work based on the AspectKE model. This chapter describes the language design of the AspectKE* programming language, which particularly features high-level program analysis predicates and functions that can provide information on future behavior of a program. Moreover it builds a secure distributed chat system with the proposed language constructs.

Chapter 7 presents the runtime system that supports the realization of the AspectKE* language. It discusses the efficient two-staged implementation strategy, the dual value evaluation mechanism, the design and use of the AspectKlava runtime Java package, and how static analysis - interprocedural data-flow analysis - is performed and integrated into the aspects.

Chapter 8 demonstrates the usability of AspectKE*, and assesses the expressiveness and performance of this language through case studies and benchmarking.

Chapter 9 re-states the major contributions of this dissertation and outlines directions of future work.

Chapter 2

Setting the Scene

In this chapter, we will present background material for better understanding this dissertation.

Section 2.1 introduces the research background together with related work, which covers work in different research domains that our dissertation draws on, including static program analysis, aspect-oriented programming for enforcing security policies, process calculi, security of coordination languages and tuple spaces. Some of this work directly inspires our work. Related work that is not suitable to discuss here but also relevant for our work is discussed at suitable places in the subsequent chapters.

Section 2.2 formally presents KLAIM [DFP98], a process calculus for modeling distributed tuple spaces with process mobility. A running example in the electronic health care setting is also demonstrated. KLAIM provides the basic building blocks for our language, as AspectKE, AspectK and AspectKE* are all essentially the designs and implementations of a aspect-oriented KLAIM model.

2.1 Research Background and Related Work

2.1.1 Static Program Analysis

Static program analysis is performed without actually executing programs, and in most cases is performed on source code or object code. It can predict safe and computable approximations to the set of values or behaviors generated dynamically when executing a program [NNH05]. Examples include to compute where values originate from and might flow to, what possible values that expressions might be evaluated to, what values might reach a certain program point of interest, etc.

Traditionally, static program analysis is used to optimize code [ASU86]. A growing number of static analyses have been used in the verification of properties for safety-critical software as well as discovery of bugs in potentially vulnerable code, e.g, [BBC+06].

The analysis provides *approximate* properties of the programs being analyzed, which are usually divided into three classes according to their analysis nature:

- *Over-approximation* captures the entire behavior of a program. It estimates the program behaviors that *may* happen along all the execution paths
- *Under-approximation* captures a subset of all possible behaviors of a program. It estimates the program behavior that *must* happen along all execution paths
- *Undecidable-approximation* can't decide whether the approximation behaviors belong to the program or not. Its result usually can not give meaningful information.

The nature of the properties determine the type of analysis technique that will be adopted, in order to get a satisfactory analysis result. In this dissertation we use the over-approximation analysis to ensure the capturing of all behaviors (intended and malicious) that *may* occur in a program.

It is important that the program analysis should be *semantics* based, which means that the information obtained from the analysis can be proved to be safe (or correct) with respect to the semantics of a programming language.

There are various types of analysis techniques for answering analysis questions on programs with different language constructs, such as Data Flow Analysis, Control Flow Analysis, Abstract Interpretation and Type systems [NNH05].

Recently, Flow Logic [NN02] has been proposed which bridges the gap between these approaches and can cope with a wide variety of programming languages as well as process calculi.

Model checking [BK08] is complementary to static analysis techniques. For a given a model, it can test whether this model meets certain specifications by performing exhaustive exploration of the possible states in a system. It is a powerful technique for precise verification of software and hardware, but suffers the so-called state explosion problem, and becomes intractable for program with large state spaces and undecidable for infinite ones. The problem can however be addressed by using abstraction techniques.

In this dissertation, we use static analysis to find potentially malicious code, and restrict ourselves to Data Flow Analysis, an analysis technique for collecting data-flow information of variables in a program. In Chapter 4, we propose a simple form of data-flow analysis, named *behavior analysis*, to illustrate the essential idea of integrating static analysis into aspect-oriented process calculus AspectKE. Chapter 7 gives a more elaborated illustration of the actual data-flow analysis developed on Java bytecode, which is used for implementing our experimental programming language AspectKE*.

2.1.2 Policy Enforcement Mechanisms

In this subsection, we relate our work to the state of the art techniques for policy enforcement. In particular, we outline existing aspect-oriented programming techniques for providing data-flow and control-flow information, and highlight our research directions.

2.1.2.1 Inlined Reference Monitors

Various techniques for enforcing security policies exist, and the most traditional one is a *reference monitor* that observes software execution and mediates dynamically all access to objects by subjects (introduced in [A+72]). Instead of mixing monitoring code in target system, Inlined Reference Monitors (IRMs) [ES00] use a load-time, trusted program rewriter to insert security code into a target application, bringing in a self-monitor application which performs security checks when it executes. There are many IRM systems implemented by various program rewriters (e.g. [UES00, ET99, BLW05, Ham06]), ensuring that different applications obey their corresponding security policies.

2.1.2.2 Aspect-oriented Programming

Independently, the aspect-oriented programming (AOP) paradigm [KHH+01] emerged and acted as another effective mechanism for tackling the same issue. The main aim of AOP is to address so-called *crosscutting concerns* that exists in traditional programming models like object-oriented programming or procedural programming. Crosscutting concerns are parts of a program which rely on or must affect many other parts of the system, which often cannot be cleanly decomposed from the rest of system, and can result in either *scattering*, i.e., code duplication, and/or *tangling*, i.e., significant dependencies between systems.

For example, Listing 2.1 shows a very simple method for transferring an amount from one account to another in a banking application.

```
1  void transfer (Account from, Account to, int amount){
2     if (from.getBalance() < amount) {
3        System.err.println("Insufficient Funds");
4     }else{
5        from.withdraw(amount);
6        to.deposit(amount);
7     }
8   }
```

Listing 2.1: Bank Transfer Function

However, this method overlooks certain concerns that a practical application would require. For example, it lacks mechanism to check whether the current user has the permission to perform this operation, and it does not contain any logging facilities to generate system log for diagnostics purpose. Listing 2.2 shows one example to address these concerns in traditional programming approach, where Lines 2-4 implement the logging facility and Lines 5-6 address the security concern.

```
1  void transfer (Account from, Account to, int amount){
2     Logger logger = new Logger ();
3     logger.info ("Transferring " + amount + " from "
4                  +from.user + " to " +to.user);
5     if (! checkUserPermission(this.user,from)){
6        System.err.println(this.user + " has no permission");
7     }else{
8        if (from.getBalance() < amount) {
9           System.err.println("Insufficient Funds");
10       }else{
11          from.withdraw(amount);
12          to.deposit(amount);
13       }
14    }
```

```
15 }
```

Listing 2.2: Bank Transfer Function with Other Concerns

As Figure 2.1 illustrated, these extra concerns not only tangle with basic transfer functionality as shown in Listing 2.2, but might also scatter across numerous other methods. Improper handling of these (crosscutting) concerns leads to loss of modularity and thus decreases the comprehensibility and maintainability of software systems.

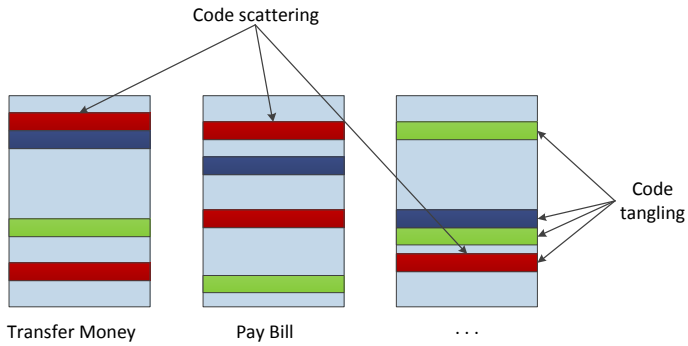

Figure 2.1: Code Scattering and Tangling

Indeed security (and logging) is naturally identified as one kind of crosscutting concern that aspect-oriented programming was designed to deal with. Instead of using a rewriter to inject monitoring code as in IRM approach, security policies are directly encapsulated in *aspects* which are automatically invoked when the target program executes certain actions. The most popular AOP language AspectJ [KHH+01] uses the *pointcut and advice* model, where the *join points* are points in a program where additional or alternative behavior can be introduced, through a *weaving* procedure, by *advice* with a *pointcut* that specifies join points of the program. The language developed in this dissertation also adopts this model.

Listings 2.3 and 2.4 present a logging aspect and a security aspect in AspectJ. They can be *weaved* into Listing 2.1 to achieve the same functionalities as Listing 2.2 does. Both aspects contain: 1.*pointcuts* (Lines 3-5 in Listing 2.3, and Lines 2-5 in Listing 2.4) that specify when and where to insert additional behaviors - in this case when invoking the transfer function; 2.*advices* (Lines 6-10 in Listing 2.3 present before advice, which shall be executed before the execution of transfer function; Lines 6-13 in Listing 2.4 present around advice, which takes over the execution of original transfer function. Notice proceed statement at Line 11 returns back to the execution of original transfer function.) This dissertation

will not present syntax of AspectJ in details, but rather introduce the core concepts of AOP.

```
1  aspect Logging {
2     Logger logger = new Logger();
3     pointcut trans(Account from,Account to, int amount):
4        call(void Bank.transfer (Account,Account,int))
5        && args(from, to, amount);
6     before(Account from,Account to, int amount):
7           trans (from,to, amount){
8        logger.info("Transferring " + amount + " from "
9                    +from.user + " to " +to.user);
10    }
11 }
```

Listing 2.3: Logging Aspect in AspectJ

```
1  aspect Security {
2     pointcut trans(Bank bank,Account from,Account to, int amount):
3        call(void Bank.transfer (Account,Account,int))
4        && args(from, to, amount)
5        && target(bank);
6     void around(Bank bank,Account from,Account to, int amount):
7              trans (bank,from,to, amount){
8        if (! Bank.checkUserPermission(bank.user,from)){
9           System.err.println(bank.user + " has no permission");
10       }else{
11          proceed (bank,from,to,amount);
12       }
13    }
14 }
```

Listing 2.4: Security Aspect in AspectJ

Compared with Figure 2.1, Figure 2.2 shows that aspect oriented programming can improve modularity of a program by implementing crosscutting concerns as aspects.

2.1.2.3 IRM, AOP and Static Analysis

There is a close connection between AOP and IRM. Hamlen and Jones [HJ08] propose an aspect-oriented security policy specification language SPoX that is enforced by IRMs, which establish a formal connection between AOP and IRMs. JavaMOP [Che05] implements IRM using AspectJ aspects as the instrumentation mechanism. Our work takes the AOP approach to internalize the reference

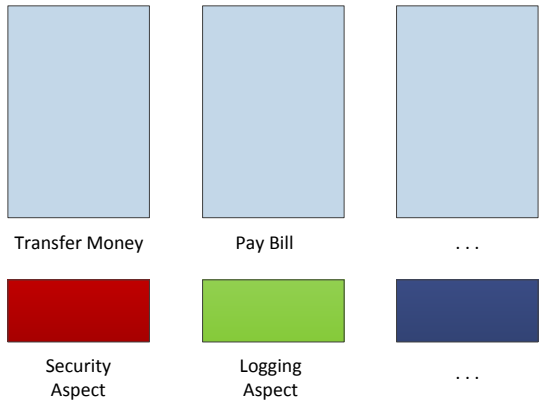

Figure 2.2: Improve Modularity with AOP

monitor for enforcing security policies to tuple space systems, and directly encodes security concerns inside aspects.

Most researches focus on the class of security policies that can be enforced by monitoring execution of a target system [Sch00]. Some research efforts are taken towards enforcing more demanding policies that cannot be easily enforced alone by using the monitoring based approach, but require combination of other techniques, for example, static analysis. Jif [MZZ+01] extends Java with a security-typed system, which supports not only access control but also information flow control. In [SMH01], the authors outline several promising methods such as IRM, type systems and certifying compilers, for enforcing more demanding security policies and also argue that synergies among these approaches will achieve remarkable results. We consider our approach – aspect-oriented programming with data-flow analysis – is a comparable research direction to IRM with type systems, for enforcing challenging security policies. In addition, our approach potentially has better modularity for defining security policies.

2.1.2.4 Data-flow and Control-flow in AOP Languages

Much work has been done in aspect-oriented programming communities, that provides various language constructs and techniques for identifying the data flow and control flow between join points. Those language constructs can serve as powerful policy enforcement mechanisms. AspectJ's cflow [KHH+01] captures the control flow between join points. dflow pointcut [MK03] identifies join points based on the data-flow information. Tracematches [AAC+05] gives advice based on the execution history of computation. However, these systems can only refer to the past and current events.

A few AOP languages propose mechanisms to trigger aspects by future control flow of a program, e.g, `pcflow` [Kic03] and `transcut` [SMH09]. However, they lack support for providing data-flow information in the future. Some advanced AOP languages (e.g., [AM07, CN04]) offer methods for referring to future behavior of a program in aspects, which in theory can be used to specify security policies depending on future control flow and data flow. However, they usually lack formal semantics and offer only access to low level (e.g., bytecode-level) information of a program, which makes it hard to understand and to develop appropriate underlying analyses for enforcing security policies. As they lack high level abstraction for presenting analysis results, it is not trivial to use the results for composing security policies.

In this dissertation, the formal semantics of a process calculus AspectKE (presented in Chapter 3 and 4) clarifies how to develop useful behavior (program) analyses to obtain future data flow of a program. AspectKE presents the analysis results through appropriate language abstraction, and formally paves the way for integrating program analysis techniques into policy specification and enforcement procedure. Chapters 6, 7 and 8 describe our experiences of building the practical prototype language AspectKE* with static analysis components included.

2.1.3 Security with Aspect-Oriented Programming

There are many papers that explore AOP techniques to enforce security policies.

One line of work directly or indirectly uses the popular Java-based general purpose AOP languages like AspectJ [KHH+01], Hyper/J [OT00], CaesarJ [AGMO06], to express and enforce security policies.

Some have reported their experiences of using the above languages for enforcing access control policies. For example, In [WVD01] the authors present general guidelines for how to compose access control aspects in AspectJ. In [VPDW+05] an enforcement of application-specific policies in an access control service is implemented in CaesarJ. Some explore the expressivity of these AOP languages regarding security policies. For example, Phung and Sands [PS08] identify classes of reference monitor-style policies that can be defined and enforced by AspectJ. They also present a method to realize some history-dependent security policies which cannot be naturally expressed in AspectJ. Some use these languages to implement classical security models. For example, Ramachandran et al. [RPW06] discusses using AspectJ for implementing multilevel security and demonstrate how aspects, in comparison with traditional programming, can guarantee better security assurance.

Some have indirectly used these languages: they first define their policies using certain policy languages with formal specification, and then translate or map them into the concrete aspects of AOP languages,instead of directly coding the policies into the aspects. Oliveira et al. [dOWKK07] use their own rewrite-based system to express access control policies and map them into an AspectJ program. In [CCBR06], availability requirements are expressed in a formal model that combines deontic and temporal logics, and then translated into availability aspects in AspectJ. One common advantage of these approaches is that policies can be formalized through security oriented languages that are naturally more suitable for security considerations than other general purpose languages. Another advantage is that some policy languages have formal semantics that enable formal verifications. Our language is designed with security in mind and has a formal semantics (AspectKE) which enables us to reason about the defined policies when needed.

Even though these well-known AOP languages have industrial strength and can be readily used for policy enforcement mechanisms, they have their limitations; it is quite difficult to apply these AOP languages to the types of systems we have been studying. For example, the languages are designed for programs that run on a local machine. And they do not natively support pointcuts for a distributed system. They also lack a pointcut mechanism to capture the future execution of a program, and thus are unable to enforce *predictive access control policies* – policies that rely on the future behavior of a program. All the above issues are explicitly addressed by our work. In particular, to the best of our knowledge few, if any, proposals have ever reported how to use aspect-oriented programming techniques to enforce secondary use of data policies, which is becoming increasingly important in large IT systems, and provide a predictive access control policy enforcement mechanism.

Some researchers design their own special purpose aspect languages or systems to study security enforcement mechanisms. For example, HarmlessAML [Dan07] is an aspect-oriented extension of the functional language Standard ML, and has a type system that guarantees well-typed harmless advice does not interfere with mainline logic computation. Our work has a different research focus on enforcing access control policies to a distributed computing model and on studying how properties obtained from behavior analyses can be used to specify access control policies. As mentioned earlier, in [MK03] the dataflow pointcut is proposed which can specify where aspects should be applied based on the origins of values in the past execution, and it is useful in situation where flow of information is important. Our work tackles the similar issue, but checks the flow of information in the future, which is particularly useful to enforce predictive access control policies.

2.1.4 Process Calculi

Process Calculus (or process algebra) is a formal approach to describe and model concurrent systems. It is a useful tool for succinctly describing interactions, communications, and synchronizations between independent processes.

A system represented with process calculi normally consists of several processes, which are recursively defined by one or more syntactic categories (including itself). These processes are essentially defined by operators working on basic primitives of a process, e.g, actions or events.

The semantics of the process calculi describe how a system, represented in a process calculus, may evolve. A common way of describing its semantics is through *Structural Operational Semantics*(SOS) [Plo81], which defines the behavior of a program in terms of transition relations. In this way one can easily consider the system defined by process calculi as a well-specified transition system. Other approaches such as denotational semantics and axiomatic semantics [NN07], primarily used in programming languages, can also specify the meaning of a process calculus. In this dissertation, we take the SOS approach to present the meaning of our calculi.

2.1.4.1 Process Calculi with Process Communication and Mobility

Classical process calculi such as CSP [Hoa78], CCS [Mil82] focus on describing synchronous communication among different processes. For example, CCS includes operators for describing parallel composition ($P|Q$, which allows computation in processes P and Q to proceed simultaneously), choice between actions ($P + Q$, either proceed with P or Q) and process handshaking (when processes $c.P$ and $\bar{c}.Q$ executed in parallel, they can synchronize their execution on communication channel c, when the handshaking succeeds they continue as P and Q).

The π-calculus [MPW92], developed from CCS, is the most influential calculus for mobile processes. It could naturally express processes that have changing structure. In π-calculus, the *name* is the central notation which can represent both *communication channels* and *variables*. The communication is performed by input prefixing $c(x).P$ (a process waiting for a message to be sent by channel c, and bound with variable x) and output prefixing $\bar{c}\langle y\rangle.Q$ (a process sending a name y on channel c). Note that the communication can only occur if both sides of the channel are ready to execute. When $c(x).P$ is executed in parallel with $\bar{c}\langle y\rangle.Q$, name y will be delivered through the channel to the receiving process, and variable x inside P will be bound with y. The process P can then use y as a channel name and, if other processes are blocked and waiting for communication

on that channel, it could trigger execution of other processes in parallel.

The KLAIM process calculus introduced below is influenced by these process calculi.

2.1.4.2 Process Calculi and Programming Languages

Calculi (or also Process Calculi) are high-level abstractions of computing models. They are often used to formally describe and clarify the essential features behind various practical programming language designs. Just as λ-calculus [Bar84] is a useful model to study theory of functional programming languages, and *typed* λ-calculus [Bar92] inspired the development of typed programming languages (such as ML or Java), etc, process calculi can formally express the core idea of a computing model, explore its properties of the concurrent computing model, and inspire the development of practical system.

Here we mention examples merely in the coordination language domain: Linda [Gel85] inspires the design of practical tuple space system JavaSpace [FAH99], and KLAIM [DFP98] is the model of X-KLAIM [BDNFP98] programming language (realized tuple space system with process mobility). We will introduce these in details below.

Again in this dissertation, we first introduce process calculus AspectKE and formally present our computing model. Then we develop the practical programming language AspectKE* based on the formal model.

2.1.5 Coordination Languages and Tuple Spaces

2.1.5.1 Tuple Space Languages

Coordination languages allow two or more parties to communicate via coordinating operations to accomplish shared goals in a network.

Linda [Gel85] was the first coordination language that is based on a shared global environment *tuple space*. Coordinating activities among several parallel processes are performed by basic asynchronous communication primitives. A tuple space is a repository of tuples that can be concurrently accessed by processes using the provided four basic primitives. These primitives enable a process to write and retrieve (through out,read/in actions) data to and from a tuple space based on pattern-matching, and to create a process for execution (through the eval action). Linda implementations can be found in programming languages such as Prolog, Ruby, Java or Lisp. The Java implementations of

Linda includes IBM's TSpaces [LCX+01] and Sun's JavaSpaces [FAH99].

KLAIM [DFP98] is tuple space based process calculi for programming distributed tuple space system that supports process mobility. It uses classical primitives from CCS and π-calculus, and additionally, integrates the primitives from classical tuple space languages such as Linda. A KLAIM program contains multiple shared tuple spaces distributed over a network, instead of a single globally shared tuple space like in Linda. Consequently, *node* is an important concept in KLAIM, which serves as an abstraction of host computer which is connected to the network that accommodates processes and a tuple space. Besides the standard actions to access tuples, a KLAIM process can create new processes on a local or remote node (through the eval action), and create a new remote node (through the newloc action). We will formally present KLAIM in Section 2.2, and use it to introduce our aspect-oriented version of KLAIM in the subsequent chapters.

KLAIM has later evolved to the KLAIM family (reviewed in [BBD+03]), including cKlaim, OpenKlaim, HotKlaim, OKlaim and X-Klaim etc. The prototype language of KLAIM is X-Klaim [BDNFP98], whose runtime system depends on Klava [BDP02]. Klava is implemented in Java and has proved to be suitable for programming many distributed applications involving code mobility. Our AspectKE* prototype language is built on top of it.

2.1.5.2 Security in Tuple Space Languages

Generally, secure shared tuple-space coordination languages can be classified into two categories in terms of the underlying access control mechanisms [FLZ06]. The entity-driven approach (additional information, associated to resources such as tuple spaces, tuples and single data fields, list the entities which are allowed to access the resources) e.g., Secure Lime [HR03] and KLAIM [DFP98]; and the knowledge-driven approach (both resources and processes are decorated with relevant additional information, and the processes can only access resources if they prove to keep those relevant additional information) e.g., SecOS [VBO03] and SecSpaces [GLZ06]. Our language is suitable for expressing access control policies that fit both an entity-driven approach and a knowledge-driven approach, as the additional information is essentially expressed in aspects and is not embedded in resources or processes. Moreover, this additional information is not limited to the past and current facts used in previous work, e.g., password [HR03], locks [VBO03] or partitions [GLZ06], but also facts about the future, e.g., how particular data will be used.

Regarding policy enforcement to KLAIM based languages, some authors use control and data-flow analyses that are written in Flow Logic approach (e.g. [HPN06,

$N \in$ **Net**	N	::=	$N_1 \,\|\|\, N_2 \;\mid\; l :: P \;\mid\; l :: \langle\, \overrightarrow{l} \,\rangle$	
$P \in$ **Proc**	P	::=	$P_1 \mid P_2 \;\mid \sum_i a_i.P_i \;\mid\; *P$	
$a \in$ **Act**	a	::=	$\mathbf{out}(\overrightarrow{\ell})@\ell \mid \mathbf{in}(\overrightarrow{\ell^\lambda})@\ell \mid \mathbf{read}(\overrightarrow{\ell^\lambda})@\ell \mid \mathbf{eval}(P)@\ell \mid \mathbf{newloc}(!u)$	
$c \in$ **Cap**	c	::=	**out** | **in** | **read** | **eval** | **newloc**	
$\ell, \ell^\lambda \in$ **Loc**	ℓ	::=	$u \;\mid\; l$	$\ell^\lambda ::= \ell \mid !u$

Table 2.1: KLAIM Syntax – Nets, Processes and Actions

HNNP08]). Others use type systems (e.g. [DFP00, DFPV00]), and [DGH+08] combine these two lines of work. They can be used to enforce very advanced security policies, however, all of them require users to explicitly annotate policies in the main code (e.g. attach policies to each location). Our approach, however, avoids this by specifying policies inside the aspects, thus achieving a better separation of concerns.

2.2 KLAIM

AspectKE, AspectK, and AspectKE* are essentially all extensions of the KLAIM (*Kernel Language for Agents Interaction and Mobility*) coordination language [DFP98] with support for aspect-oriented programming. In this section we will review the fragments of KLAIM used in the following chapters.

KLAIM is a language specifically designed to program distributed systems with mobile components that interact with each other on multiple distributed tuple spaces (databases). KLAIM uses a Linda-like generative communication model. However, instead of a global shared tuple space (shared database), KLAIM associates each node with a local tuple space. Each node can also have processes associated with it. The KLAIM computing primitives allow programmers to distribute and retrieve data and processes to and from locations (nodes) of a net, evaluate processes at remote locations and introduce new locations to the net.

2.2.1 Syntax of KLAIM

The syntax of a KLAIM fragment is displayed in Table 2.1.

A net (in **Net**) is a parallel composition of located processes and/or located

tuples. For simplicity, components of tuples can be location constants only[1]. Nets must be *closed*: all variables must be in the scope of a defining occurrence (indicated by an exclamation mark).

A process (in **Proc**) can be a parallel composition of processes, a guarded sum of action prefixed processes, or a replicated process (indicated by the $*$ operator). We write $\mathbf{0}$ for a nullary sum, $a.P$ for a unary sum, and $a_1.P_1 + a_2.P_2$ for a binary sum.

An action (in **Act**) operates on locations, tuples and processes. A tuple can be output to, input from (read and delete the source) and read from (read and keep the source) a location. Processes can be spawned at a location. New locations can also be created. The actual operation performed by an action is called a *capability* (in **Cap**) – this is a key concept when formalizing uses of data in later chapters. We do not distinguish real locations and data: all of them are called locations (in **Loc**) in our setting, which can be location constants l, defining (i.e. binding) occurrences of location variables $!u$ (where the scope is the entire process to the right of the occurrence), and use of location variables u.

Well-Formedness of Locations and Actions To express the well-formedness conditions we introduce the functions bv and fv for calculating the bound and free variables of the various kinds of locations that may occur in actions. The definitions are standard, in particular, $bv(l, u, !v) = \{v\}$ and $fv(l, u, !v) = \{u\}$. In later chapters we will also make use of another function, lc, to extract the location constants appearing in an action.

An input action (and a read action) is well-formed if its sequence $\overrightarrow{\ell^{\lambda}} = \ell_1, \cdots, \ell_k$ (for $k \geq 0$) of locations is well-formed. This is the case when the following two conditions are fulfilled:

$$\forall i, j \in \{1, \cdots, k\} : i \neq j \Rightarrow bv(\ell_i^{\lambda}) \cap bv(\ell_j^{\lambda}) = \emptyset \text{ and}$$
$$bv(\overrightarrow{\ell^{\lambda}}) \cap fv(\overrightarrow{\ell^{\lambda}}) = \emptyset$$

The first condition demands that we do not use multiple defining occurrences of the same variable in an action. The second condition requires that bound variables and free variables cannot share the same name in a single action. Thus we disallow $\mathbf{in}(!u, !u)@l$ as well as $\mathbf{in}(!u, u)@l$.

We do not impose further restrictions on the syntax of output action, the process evaluation action and the location creation action.

[1]Compared with the original KLAIM, we do not allow processes to be components of tuples.

$$l :: (P_1 \mid P_2) \equiv (l :: P_1) \,||\, (l :: P_2) \qquad l :: (*P) \equiv l :: (P \mid *P)$$

$$\frac{N_1 \equiv N_2}{N \,||\, N_1 \equiv N \,||\, N_2}$$

Table 2.2: KLAIM Structural Congruence

2.2.2 Semantics of KLAIM

Informally the meaning of a KLAIM program is as follows:

1. a node is selected for the next step of execution
2. if the process at the node is a choice, then one of the enabled choices is chosen non-deterministically and executed as described in the following four steps
3. if the prefix of the process is an output action, the output is performed
4. if the prefix of the process is an input (either destructive or non-destructive), the input action is enabled if there is a matching tuple at the target location, and the input is performed and appropriate variables are bound in the remainder of the process
5. if the prefix is an eval, the process is spawned at the target location
6. if the prefix is a newloc, the network is dynamically extended with a new location and the continuation process is given the address of that location
7. then return to Step 1

Notice that we do not need to deal with parallelism and replication within nodes because, at the cost of having duplicate addresses in the network, these can be lifted to the net level.

More formally, the semantics is given by an one-step reduction relation on nets and is defined in Table 2.3. We make use of a structural congruence on nets; this is an associative and commutative (with respect to ||) equivalence relation and the interesting cases are defined in Table 2.2. The semantics also assumes that all location constants referred in a Klaim program do exist, and a static check is performed before execution to guarantee this property. Notice this is a slightly simplified way to formulate the semantics rules originally presented at [DFP98], where it differentiates real sites from logical locations and will check

$l_s :: (\mathbf{out}(\overrightarrow{l})@l_0.P + \cdots) \to l_s :: P \mid\mid l_0 :: \langle \overrightarrow{l} \rangle$	
$l_s :: (\mathbf{in}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots) \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \to l_s :: P\theta$	if $match(\overrightarrow{\ell^\lambda}; \overrightarrow{l}) = \theta$
$l_s :: (\mathbf{read}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots) \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \to l_s :: P\theta \mid\mid l_0 :: \langle \overrightarrow{l} \rangle$	if $match(\overrightarrow{\ell^\lambda}; \overrightarrow{l}) = \theta$
$l_s :: (\mathbf{eval}(P')@l_0.P + \cdots) \to l_s :: P \mid\mid l_0 :: P'$	
$l_s :: (\mathbf{newloc}(!u).P + \cdots) \to l_s :: P[l'/u] \mid\mid l' :: \mathbf{0}$	with l' fresh

$$\frac{N_1 \to N_1'}{N_1 \mid\mid N_2 \to N_1' \mid\mid N_2} \qquad \frac{N \equiv N' \quad N' \to N'' \quad N'' \equiv N'''}{N \to N'''}$$

Table 2.3: KLAIM Reaction Semantics (on closed nets)

$match(\, !u, \overrightarrow{\ell^\lambda}\, ;\, l, \overrightarrow{l}\,) = [l/u] \circ match(\, \overrightarrow{\ell^\lambda}\, ;\, \overrightarrow{l}\,)$	$match(\, \epsilon\, ;\, \epsilon\,) = id$
$match(\, l, \overrightarrow{\ell^\lambda}\, ;\, l, \overrightarrow{l}\,) = match(\, \overrightarrow{\ell^\lambda}\, ;\, \overrightarrow{l}\,)$	$match(\, .\, ;\, .\,) = \mathsf{fail}$ otherwise

Table 2.4: Matching Input Patterns to Data

dynamically whether a logical location (refereed in a Klaim program) associates with an existing site.

The rule for **out** is rather straightforward; it uses the fact that the action selected may be part of a guarded sum to dispense with any other alternatives. The rules for **in** and **read** only progress if the formal parameters $\overrightarrow{\ell^\lambda}$ match the candidate tuple $\overrightarrow{l}$. The details of the matching operation are given in Table 2.4 (explained below). If the matching succeeds and produces a substitution then the rule applies; if no substitution is produced (due to a fail in part of the computation) then the rule does not apply. The rule for **eval** will spawn a new process at a specified location before continuing with the following process P. The rule for **newloc** will create a fresh empty location and substitute it for u in the continuation process P.

The matching operation of Table 2.4 returns a substitution θ being a (potentially empty) list of pairs of the form $[l/u]$; if the list is empty, it is denoted by id. Notice that the definition does not apply to location variables because tuples in the tuple space may only contain location constants and the reaction semantics is restricted to closed nets.

2.2.3 Running Example

Health Care Information Systems are gradually becoming prevalent and indispensable to our society. An *electronic health record* (EHR), part of a system's database, stores a patient's data and is created, developed, and maintained by the health care providers.

To illustrate the use of KLAIM, we now introduce a typical EHR system, which is inspired by [EB04], and the scenario presented here is used throughout the dissertation.

The EHR database (EHDB) stores all patient healthcare records and we assume that there are two types of data for each patient: medical records (MedicalRecord) and private notes (PrivateNote). Medical records are entries created by doctors and so are the private notes. However, the latter are of a more confidential nature. Also we distinguish between past records (Past) that have been entered into the EHR system previously and recent records (Recent) that have been created since the patient was admitted to the hospital. We therefore assume that the EHR database contains tuples with the following five fields:

patient	The name of the patient
recordtype	The type of record: MedicalRecord or PrivateNote
author	The author of the record
createdtime	The time of creation of the record: Recent or Past
subject	The record's content

For example ⟨Alice, MedicalRecord, DrSmith, Recent, text⟩ is a recent medical record of Alice, created by DrSmith and it has content text.

Doctors and nurses, as well as the patient, can access a patient's record. We model these actors as locations in a network. The process at the location represents the actions of an individual and the data is the individual's local "knowledge". As an example, the following process expresses that DrSmith reads one of the Past medical records for Alice created by DrHansen before she was admitted to this hospital, writes some of the information in her own note (in location DrSmith) and then creates a new medical record for the patient:

$$\begin{aligned} \mathsf{DrSmith} :: \quad & \mathbf{read}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, !\mathit{content})@\mathsf{EHDB}. \\ & \mathbf{out}(\mathsf{Alice}, \mathit{content})@\mathsf{DrSmith}. \\ & \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB} \end{aligned}$$

Here DrSmith will first consult location EHDB and read a five-tuple whose first four components are Alice,MedicalRecord,DrHansen, and Past respectively and the corresponding fifth component is assigned to variable *content*. The second action will write the *content* read at the first action to the location associated with DrSmith. The final construct will write a new five-tuple to location EHDB

for this patient whose last three components indicate that the author is DrSmith, it is a Recent medical record and the content is newtext.

To illustrate the semantics of KLAIM let us consider the following net, consisting of locations EHDB and DrSmith:

$$
\begin{array}{ll}
 & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\
\| & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\
\| & \mathsf{DrSmith} :: \mathbf{read}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, !\mathit{content})@\mathsf{EHDB}. \\
 & \qquad \mathbf{out}(\mathsf{Alice}, \mathit{content})@\mathsf{DrSmith}. \\
 & \qquad \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB}
\end{array}
$$

The execution may proceed as follows:

$$
\begin{array}{ll}
 & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\
\| & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\
\| & \mathsf{DrSmith} :: \mathbf{read}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, !\mathit{content})@\mathsf{EHDB}. \\
 & \qquad \mathbf{out}(\mathsf{Alice}, \mathit{content})@\mathsf{DrSmith}. \\
 & \qquad \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB} \\
\rightarrow & \\
 & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\
\| & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\
\| & \mathsf{DrSmith} :: \mathbf{out}(\mathsf{Alice}, \mathsf{alicetext})@\mathsf{DrSmith}. \\
 & \qquad \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB} \\
\rightarrow & \\
 & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\
\| & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\
\| & \mathsf{DrSmith} :: \langle \mathsf{Alice}, \mathsf{alicetext} \rangle \\
\| & \mathsf{DrSmith} :: \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB} \\
\rightarrow & \\
 & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\
\| & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext} \rangle \\
\| & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\
\| & \mathsf{DrSmith} :: \langle \mathsf{Alice}, \mathsf{alicetext} \rangle
\end{array}
$$

DrSmith first reads the tuple $\langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle$ from EHDB; the binding of the variable *content* is reflected in the continuation of the process. In the second step DrSmith outputs a tuple that consists of Alice together with a bound *content* (alicetext) to her own tuple space. In the final step, a new tuple that represents a new medical record is written to location EHDB.

2.3 Concluding Remarks

Our research work does not happen in a vacuum. This chapter presented the essential research background and related work that our research directly builds upon, and some of this work challenges and inspires our research. We also

presented the KLAIM process calculus, from which we designed AspectKE, AspectK and AspectKE*. They will be discussed in the following chapters.

Chapter 3

AspectKE: Trapping Actions

In this Chapter, we show how to integrate aspects into KLAIM by presenting the basic features of AspectKE (Section 3.1 and 3.2), particularly on how aspects trap actions (except the **eval** action) in a KLAIM program. We leave the advanced features of AspectKE, i.e., how aspects trap processes of a KLAIM program, to Chapter 4. In the last section of this chapter (Section 3.3), we explore the expressiveness of the introduced language constructs by enforcing various classical access control models (mandatory access control, discretionary access control and role based access control) and advice for retrofitting policies to an evolving system. This is introduced by the electronic health care setting introduced in previous chapter, Section 2.2.3.

In AspectKE, we consider a global set of aspects. The base semantics is that of KLAIM (Section 2.2). However, before performing any action, we check first if any aspect applies to the action and combine the advice of all applicable aspects before executing it (all actions in a KLAIM program are potential join points). An advice is either that the action be allowed to proceed or not. We resolve possible conflicts by ensuring that any aspect that disallows an action has priority. In the formal semantics, aspects are applied in definition order but, because aspects can only allow or disallow the join point to proceed, the order is actually immaterial.

$S \in$ **System**	S	::=	**let** $\overrightarrow{asp}$ **in** N
$asp \in$ **Asp**	asp	::=	$A[cut] \triangleq body$
$body \in$ **Advice**	$body$	::=	**case** $(cond)$ $sbody$; $body$ | $sbody$
	$sbody$	::=	**break** | **proceed**
$cut \in$ **Cut**	cut	::=	$\ell :: ca$
$ca \in$ **CAct**	ca	::=	$\mathbf{out}(\overrightarrow{\ell_t})@\ell$ | $\mathbf{in}(\overrightarrow{\ell_t^\lambda})@\ell$ | $\mathbf{read}(\overrightarrow{\ell_t^\lambda})@\ell$ | $\mathbf{newloc}(\_)$
$cond \in$ **BExp**	$cond$	::=	$\ell_1 = \ell_2$ | $cond_1 \wedge cond_2$ | $cond_1 \vee cond_2$ | $\neg\, cond$
		|	$\mathbf{test}(\overrightarrow{\ell_t})@\ell$ | $\exists u \in set : cond$ | $\forall u \in set : cond$
$set \in$ **Set**	set	::=	$\{\ell\}$ | $set \cap set$ | $set \cup set$
$\ell_t, \ell_t^\lambda \in$ **Loc**	ℓ_t	::=	ℓ | _ $\qquad\qquad$ ℓ_t^λ ::= ℓ^λ | _

Table 3.1: AspectKE Syntax - Aspects for Trapping Actions

3.1 Syntax

The Syntax of AspectKE is given by Tables 3.1 and 2.1 (the KLAIM syntax).

Table 3.1 introduces a system S (in **System**) that consists of a net N and a sequence of global aspect declarations $\overrightarrow{asp}$. An aspect declaration (in **Asp**) takes the form $A[cut] \triangleq body$: A is the aspect name, and $body$ (in **Advice**) is the advice to the trapped action. Each action (the **Act** in Table 2.1) is a potential join point that can be intercepted by AspectKE's pointcut (in **Cut**).

Moreover, _ is introduced as a don't-care parameter in the cut version of actions, and in the **test** primitive of conditional expressions (**BExp**). It can match any type of location used in the program. Note in the cut, the occurrence of $!u$ and u have different meanings from those of KLAIM, which do not respectively represent defining occurrences of location variable and use of location variable. Rather they bind different type of locations in the join point. The exact meaning of _, $!u$ and u mentioned above will be clarified when we explain the semantics of the language.

Each aspect gives a unique run-time suggestion (either **break** or **proceed**) which may depend on the evaluation of a conditional expression. The suggestion **break** suppresses the trapped action whilst **proceed** allows the trapped action to be executed. In case that multiple aspects trap an action, **break** takes precedence over **proceed**. The primitive $\mathbf{test}(\overrightarrow{\ell_t})@\ell$ evaluates to **tt** if a tuple exists in the tuple space of ℓ which matches $\overrightarrow{\ell_t}$. Besides basic boolean expressions, condition *cond* also includes *bounded* existential quantification and

universal quantification – this allows simple queries to the databases occurring in the nets.

In contrast to other aspect languages, the condition is part of the advice instead of being part of the pointcut (being evaluated before intercepting a join point). Evaluating the condition after intercepting a join point allows a more natural modelling of security policies.

Well-formedness of Cuts We define $cl(cut)$ as the list of location entities (constants as well as variables) involved in a cut. For example:

$$cl(l_s :: \mathbf{in}(!x, y)@l_0) = \langle l_s, x, y, l_0 \rangle$$

In addition to the well-formedness conditions for KLAIM, we require that the variables of $cl(cut)$ are pairwise distinct. We shall also impose that aspects are *closed*: any free variable in the body is defined in the cut. Additionally, when $!u$ is used in a cut pattern, no use of u will be allowed inside tests.

Example 3.1 To illustrate how aspects can be composed in AspectKE that work with the KLAIM program, this example shows a simple aspect that gives advice to the running example in Section 2.2.3.

$$\begin{aligned} &\mathsf{A}_1^{out}[user :: \mathbf{out}(\_, data)@\mathsf{DrSmith}] \\ &\quad \triangleq \mathbf{case}(data = \mathsf{alicetext}) \\ &\qquad \mathbf{break}; \\ &\qquad \mathbf{proceed} \end{aligned}$$

The exact meaning of this aspect is clarified in the semantics subsection, but the basic function is: trap an **out** action of processes running at location DrSmith that attempt to send a tuple with two fields. If the actual value of the second field is equal to alicetext, the aspect will break the execution of the action and its continuation process. Otherwise, the action continues. □

3.2 Semantics

The semantics is given by a one-step reduction relation on well-formed systems, nets and actions. As before, we make use of the structural congruence on nets which is defined in Table 2.2. In addition, we also re-use the operation *match* in Table 2.4, for matching input patterns to actual data.

The reaction rules are defined in Table 3.2, where $\overrightarrow{asp}$ is a global environment of aspects. The rules for systems and nets are straightforward. We discuss the rules for actions: they all make use of the function Φ for determining whether or

$$\frac{\overrightarrow{asp} \vdash N \to \overrightarrow{asp} \vdash N'}{\textbf{let } \overrightarrow{asp} \textbf{ in } N \to \textbf{let } \overrightarrow{asp} \textbf{ in } N'}$$

$$\frac{\overrightarrow{asp} \vdash N_1 \to \overrightarrow{asp} \vdash N_1'}{\overrightarrow{asp} \vdash N_1 \mid\mid N_2 \to \overrightarrow{asp} \vdash N_1' \mid\mid N_2} \qquad \frac{N \equiv M \quad \overrightarrow{asp} \vdash M \to \overrightarrow{asp} \vdash M' \quad M' \equiv N'}{\overrightarrow{asp} \vdash N \to \overrightarrow{asp} \vdash N'}$$

$$\overrightarrow{asp} \vdash l_s :: (\textbf{out}(\overrightarrow{l})@l_0.P + \cdots) \to \begin{cases} \overrightarrow{asp} \vdash l_s :: P \mid\mid l_0 :: \langle \overrightarrow{l} \rangle & \text{if } \Phi^{\overrightarrow{asp}}(l_s :: \textbf{out}(\overrightarrow{l})@l_0.P) \\ \overrightarrow{asp} \vdash l_s :: \mathbf{0} & \text{if } \neg\Phi^{\overrightarrow{asp}}(l_s :: \textbf{out}(\overrightarrow{l})@l_0.P) \end{cases}$$

$$\overrightarrow{asp} \vdash l_s :: (\textbf{in}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots) \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \to \begin{cases} \overrightarrow{asp} \vdash l_s :: P\theta & \text{if } \Phi^{\overrightarrow{asp}}(l_s :: \textbf{in}(\overrightarrow{\ell^\lambda})@l_0.P) \\ & \quad \wedge\, match(\overrightarrow{\ell^\lambda}; \overrightarrow{l}) = \theta \\ \overrightarrow{asp} \vdash l_s :: \mathbf{0} \mid\mid l_0 :: \langle \overrightarrow{l} \rangle & \text{if } \neg\Phi^{\overrightarrow{asp}}(l_s :: \textbf{in}(\overrightarrow{\ell^\lambda})@l_0.P) \end{cases}$$

$$\overrightarrow{asp} \vdash l_s :: (\textbf{read}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots) \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \to \begin{cases} \overrightarrow{asp} \vdash l_s :: P\theta \mid\mid l_0 :: \langle \overrightarrow{l} \rangle & \text{if } \Phi^{\overrightarrow{asp}}(l_s :: \textbf{read}(\overrightarrow{\ell^\lambda})@l_0.P) \\ & \quad \wedge\, match(\overrightarrow{\ell^\lambda}; \overrightarrow{l}) = \theta \\ \overrightarrow{asp} \vdash l_s :: \mathbf{0} \mid\mid l_0 :: \langle \overrightarrow{l} \rangle & \text{if } \neg\Phi^{\overrightarrow{asp}}(l_s :: \textbf{read}(\overrightarrow{\ell^\lambda})@l_0.P) \end{cases}$$

$$\overrightarrow{asp} \vdash l_s :: (\textbf{newloc}(!u).P + \cdots) \to \begin{cases} \overrightarrow{asp} \vdash l_s :: P[l'/u] \mid\mid l' :: \mathbf{0} & \text{with } l' \text{ fresh} \\ & \text{if } \Phi^{\overrightarrow{asp}}(l_s :: \textbf{newloc}(!u).P) \\ \overrightarrow{asp} \vdash l_s :: \mathbf{0} & \text{if } \neg\Phi^{\overrightarrow{asp}}(l_s :: \textbf{newloc}(!u).P) \end{cases}$$

Table 3.2: Reaction Semantics of AspectKE (on closed nets)

$$
\begin{array}{rcl}
\Phi^{A[cut]\triangleq body,\overrightarrow{asp}}(l :: a.P) & = & \text{case } check(\, extract(cut)\,;\, extract(l :: a)) \text{ of} \\
 & & \quad \mathsf{fail}: \quad \Phi^{\overrightarrow{asp}}(l :: a.P) \\
 & & \quad \theta: \quad (\mathbf{proceed} = [\![body\,\theta]\!]) \wedge \Phi^{\overrightarrow{asp}}(l :: a.P) \\
\Phi^{\varepsilon}(l :: a.P) & = & \mathbf{tt}
\end{array}
$$

Table 3.3: Trapping Aspects in AspectKE

$$
\begin{array}{l}
check(\,\alpha, \overrightarrow{\alpha}\,;\, \alpha', \overrightarrow{\alpha'}\,) = do(\alpha;\alpha') \circ check(\,\overrightarrow{\alpha}\,;\,\overrightarrow{\alpha'}\,) \\
check(\,\epsilon\,;\,\epsilon\,) = id \\
check(\,.\,;\,.\,) = \mathsf{fail} \text{ otherwise}
\end{array}
\qquad
\begin{array}{l}
do(\,u\,;\,l\,) = [l/u] \\
do(\,!u\,;\,!u'\,) = [u'/u] \\
do(\,\_\,;\,l\,) = id \\
do(\,\_\,;\,!u\,) = id \\
do(\,l\,;\,l\,) = id \\
do(\,c\,;\,c\,) = id \\
do(\,.\,;\,.\,) = \mathsf{fail} \text{ otherwise}
\end{array}
$$

Table 3.4: Checking Formals against Actuals

not all applicable aspects allow the action to **proceed** (in this case it evaluates to **tt**) or whether at least one aspect requires the action to **break** (in this case it evaluates to **ff**). Note that we have omitted **eval** from AspectKE at this point, it will be re-introduced in Chapter 4. Subject to the definition of Φ the rules and axioms should be straightforward: if Φ returns **tt** then the action will be executed; otherwise it and its continuation will be replaced with the **0** process and execution of this thread terminates. As in KLAIM, **out** simply puts tuple $\overrightarrow{l}$ into location l_0 and continues with the continuation process P; **in** and **read** only progress if the formal parameters $\overrightarrow{\ell^{\lambda}}$ match $\overrightarrow{l}$, and this operation is defined in Table 2.4. The **newloc** action creates a fresh empty location.

The function Φ is defined in Table 3.3 and makes use of three auxiliary functions. The function *check*, defined in Table 3.4, checks the applicability of each aspect in the aspect environment, and produces the corresponding bindings of actual parameters of actions to the formal parameters of advice. The auxiliary function *do* matches parameters of the pointcut against parameters of the join point; notice that a plain variable in a pointcut can only match an actual location and banged (!) variables in the pointcut can only match against binding occurrences of variables, while the don't-care (_) can match both in the join point. The last case of function *do* expresses that the function will return fail for all other combinations. For example, the attempt to match a banged variable $!u$ in the pointcut to an actual location l from the join point will return fail.

The function *extract* facilitates the checking process by producing a list of

names: the location where the trapped action occurs; the capability (**out**, **in**, **read** or **newloc**); the parameters of the action; the target location of the action. As an example,

$$extract(l :: \mathbf{out}(\ell_1, \cdots, \ell_n)@\ell') = (l, \mathbf{out}, \ell_1, \cdots, \ell_n, \ell')$$

The evaluation of function *check* relies on the evaluation of several invocations of *do* that try to match every parameter in the pointcut against the corresponding parameter in the join point. If at least one mismatch occurs, *check* will return fail, which means that this aspect will not apply to the action and we shall evaluate the next aspect against this action; otherwise, it returns a substitution, θ, which is applied to the advice (*body* θ) and we check whether the semantics of the advice is **proceed**, and continue searching through the remaining aspects, taking the conjunction of all results. In this way **break** takes precedence over **proceed**. The function $[\![\ ]\!]$ evaluates its parameter (a case statement) in the expected way. One might wonder whether the parameter (*body* θ) might contain free variables since θ may map a variable to a variable; however, the well-formedness of cuts guarantees that this does not happen since all aspects are closed. One may also note that the order in which the aspects are listed in $\overrightarrow{asp}$ does not matter.

Another option for defining the semantics of **newloc** action is:

$$\rightarrow \begin{cases} \mathbf{let}\ \overrightarrow{asp}\ \mathbf{in}\ l_s :: (\mathbf{newloc}(!u, asp).P + ...) \mid\mid N & \\ \mathbf{let}\ \overrightarrow{asp}, (asp[l'/u])\ \mathbf{in}\ l_s :: P[l'/u] \mid\mid l' :: \mathbf{0} \mid\mid N & \text{with } l' \text{ fresh} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{if } \Phi^{\overrightarrow{asp}}(l_s :: \mathbf{newloc}(!u).P) \\ \mathbf{let}\ \overrightarrow{asp}\ \mathbf{in}\ l_s :: \mathbf{0} \mid\mid N & \text{if } \neg\Phi^{\overrightarrow{asp}}(l_s :: \mathbf{newloc}(!u).P) \end{cases}$$

In this approach, we can also introduce a new aspect *asp* to the current global aspect environment. The variable u of the new aspect can be instantiated with the new location l'. However, this approach violates the *oblivious* spirit [FF05] of aspect-oriented programming: when programming the basic functionality, users should not consider defining aspects at the same time.

Example 3.2 Suppose we have a system that contains the same net as in running example of Section 2.2.3 and aspect A_1^{out} in Example 3.1:

$$\begin{array}{ll} & \mathsf{A}_1^{out}[user :: \mathbf{out}(\_, data)@\mathsf{DrSmith}] \\ \mathbf{let} & \quad \triangleq \mathbf{case}(data = \mathsf{alicetext}) \\ & \qquad \mathbf{break}; \\ & \qquad \mathbf{proceed} \end{array} \quad \mathbf{in}$$

$$\begin{array}{ll} & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\ \mid\mid & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\ \mid\mid & \mathsf{DrSmith} :: \mathbf{read}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, !content)@\mathsf{EHDB}. \\ & \qquad \mathbf{out}(\mathsf{Alice}, content)@\mathsf{DrSmith}. \\ & \qquad \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB} \end{array}$$

and some steps of execution (omitting the aspect definition):

$$
\begin{array}{ll}
 & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\
\| & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\
\| & \mathsf{DrSmith} :: \ \mathbf{read}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, !\mathit{content})@\mathsf{EHDB}. \\
 & \qquad \mathbf{out}(\mathsf{Alice}, \mathit{content})@\mathsf{DrSmith}. \\
 & \qquad \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB} \\
\rightarrow & \\
 & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\
\| & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\
\| & \mathsf{DrSmith} :: \ \mathbf{out}(\mathsf{Alice}, \mathsf{alicetext})@\mathsf{DrSmith}. \\
 & \qquad \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB} \\
\rightarrow & \\
 & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\
\| & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\
\| & \mathsf{DrSmith} :: \mathbf{0}
\end{array}
$$

Aspect A_1^{out} does not trap the **read** action, thus the **read** action executes and binds *content* with alicetext. But A_1^{out} traps the first **out** action, and the result substitution is

$$[\mathsf{DrSmith}/\mathit{user}, \mathsf{alicetext}/\mathit{data}]$$

and the case condition $\mathit{data} = \mathsf{alicetext}$ evaluates to **tt**, thus the aspect breaks the execution of this action and its continuation process. □

3.3 Advice for Access Control Models

To evaluate the expressiveness of the language and show its language features, we now demonstrate how AspectKE can be used to enforce access control policies by utilizing three well-known access control models, namely discretionary access control (Section 3.3.1), mandatory access control (Section 3.3.2) and role-based access control (Section 3.3.3), and how AspectKE can introduce new aspects for retrofitting new policies to existing systems (Section 3.3.4).

Since patient confidentiality is an important issue in the health care industry it is imperative that EHRs are protected [Win05]. To help achieve this goal, governments define many types of security policies, encapsulated in various acts and guides (e.g. [Dep03a, Dep03b]). Throughout the paper, we will enforce several security policies for the EHR system that was introduced in Section 2.2.3 and this shows different features of the language.

The first is a primary use of data policy inspired by [EB04] which regulates the basic access control concerning the *read* and *write* rights owned by doctors and nurses:

> *Doctors can read all patients' medical records and private notes, while nurses can read all patients' medical records but cannot read any private notes. Medical records and private notes can only be created by doctors.*

For simplicity, here we restrict ourselves to only focussing on **read, in** and **out** actions, while **eval** and **newloc** actions will be discussed further in Chapter 4 when enforcing other security policies.

3.3.1 Discretionary Access Control

We will show how to enforce the above policy with discretionary access control (DAC), which is a type of access control as a means of restricting access to objects based on the identity of subjects and/or the groups to which they belong [QZW+85]. We do so by using an *access control matrix* containing triples (s, o, c), identifying which subjects s can perform which operations c on which objects o. If we use the KLAIM programming model, we should equip the semantics of KLAIM with a reference monitor that consults the access control matrix when an action is executed, to check if the action is permitted. In AspectKE we can directly use aspects to elegantly *inline* the reference monitor to enforce this discretionary access control policy.

Example 3.3 The access control matrix is stored in location DAC, which contains tuples: $\langle user, recordtype, capability \rangle$. For example, if DrSmith is a doctor and NsOlsen is a nurse, then DAC might contain the following tuples:

$\langle$DrSmith, MedicalRecord, read$\rangle$
$\langle$DrSmith, PrivateNote, read$\rangle$
$\langle$DrSmith, MedicalRecord, out$\rangle$
$\langle$DrSmith, PrivateNote, out$\rangle$
$\langle$NsOlsen, MedicalRecord, read$\rangle$

We also assume that the location DAC can only be modified by privileged users, thus doctors and nurses cannot perform any **in** and **out** action on it. This can be enforced by other aspects but we omit them here.

The following aspect declarations will impose the desired requirements.

$$\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{p1_{A1}}}[user :: \mathbf{read}(_, recordtype, _, _, _)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, recordtype, \mathsf{read})@\mathsf{DAC}) \\
&\qquad \mathbf{proceed}; \\
&\quad \mathbf{break} \\
&\mathsf{A}^{in}_{\mathsf{p1_{A2}}}[user :: \mathbf{in}(_, recordtype, _, _, _)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, recordtype, \mathsf{in})@\mathsf{DAC}) \\
&\qquad \mathbf{proceed}; \\
&\quad \mathbf{break} \\
&\mathsf{A}^{out}_{\mathsf{p1_{A3}}}[user :: \mathbf{out}(_, recordtype, _, _, _)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, recordtype, \mathsf{out})@\mathsf{DAC}) \\
&\qquad \mathbf{proceed}; \\
&\quad \mathbf{break}
\end{aligned}$$

Aspects $\mathsf{A}^{read}_{\mathsf{p1_{A1}}}$, $\mathsf{A}^{in}_{\mathsf{p1_{A2}}}$, and $\mathsf{A}^{out}_{\mathsf{p1_{A3}}}$ enforce the above policy by using DAC, where the access rights for each user are described. Note that the second field of the tuple operated by these cut actions is *recordtype*, which trap an action that clearly specifies a concrete record type.

Consider the following KLAIM program that is a variant of the running example in Section 2.2.3 (in that the user is nurse NsOlsen instead of doctor DrSmith) and is equipped with the above four aspects:

$$\begin{aligned}
\mathsf{NsOlsen} :: \quad & \mathbf{read}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, !content)@\mathsf{EHDB}. \\
& \mathbf{out}(\mathsf{Alice}, content)@\mathsf{NsOlsen}. \\
& \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{NsOlsen}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB}
\end{aligned}$$

The first **read** action will be trapped by aspect $\mathsf{A}^{read}_{\mathsf{p1_{A1}}}$, and the resulting substitution is

$$[\mathsf{NsOlsen}/user, \mathsf{MedicalRecord}/recordtype]$$

and the condition **test**(NsOlsen, MedicalRecord, read)@DAC is evaluated. Since NsOlsen has the appropriate right according to DAC we proceed and perform this **read** action thereby giving rise to the binding of *content* to alicetext.

The second action will not be trapped by any of the aspects, so it will simply be performed and the tuple $\langle \mathsf{Alice}, \mathsf{alicetext} \rangle$ is output to location NsOlsen.

The last action will be trapped by aspect $\mathsf{A}^{out}_{\mathsf{p1_{A3}}}$ and after the substitution we evaluate the condition **test**(NsOlsen, MedicalRecord, out)@DAC which is evaluated to **ff** and thus we **break** the execution.

However, the KLAIM program can also execute **read** or **in** actions without specifying the record type, e.g., using !*recordtype* instead of *recordtype*, users can thus get a record as follows:

$$\mathsf{NsOlsen} :: \mathbf{read}(\mathsf{Alice}, !recordtype, \mathsf{DrHansen}, \mathsf{Past}, !content)@\mathsf{EHDB}$$

where a successful input action can retrieve any type of EHR record.

None of the above aspects can trap these input actions, thus we have to enforce additional aspects so that the above input actions will not bypass our aspects and consequently break the policy. The simple aspects forbid any attempts to **read** or **in** (read and then remove) EHR records without specifying the record type:

$$\mathsf{A}^{read}_{\mathsf{p1}_{\mathsf{A4}}}[user :: \mathbf{read}(\_, !recordtype, \_, \_, \_)@\mathsf{EHDB}] \triangleq \mathbf{break}$$

$$\mathsf{A}^{in}_{\mathsf{p1}_{\mathsf{A5}}}[user :: \mathbf{in}(\_, !recordtype, \_, \_, \_)@\mathsf{EHDB}] \triangleq \mathbf{break}$$

One may wonder why not build the above two aspects on top of aspects $\mathsf{A}^{read}_{\mathsf{p1}_{\mathsf{A1}}}$ and $\mathsf{A}^{in}_{\mathsf{p1}_{\mathsf{A2}}}$ by directly replacing *recordtype* with !*recordtype* in their pointcut, respectively. The reason is that these aspects will not be well-formed: when trapping actions, *recordtype* binds with a variable, which cannot be used in a test condition such as $\mathbf{test}(user, recordtype, \mathsf{read})@\mathsf{DAC}$. □

3.3.2 Mandatory Access Control

In this subsection we will show how to enforce the above policies by using mandatory access control (MAC), which is a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects [QZW+85]. Before enforcing the above policy, we first impose a comparable classical MAC policy - the Bell-LaPadula security policy [Gol99] based on a mandatory access control model. Later we enforce the above policy as a variant of the Bell-LaPadula policy. In the presentation, *security levels* are assigned to subjects (as clearances) and objects (as labels).

Example 3.4 In this scenario, we just need two security levels, and may assign security levels to subjects as follows: doctors have level high and nurses have level low; similarly we may assign objects as follows: private notes have level high and medical records have level low.

To model this policy we need to introduce a location MAC that stores tuples of the form: $\langle user, securitylevel\rangle$ and $\langle recordtype, securitylevel\rangle$. Continuing Example 3.3, we create the location MAC with the tuples:

$\langle \mathsf{DrSmith}, \mathsf{high}\rangle$
$\langle \mathsf{NsOlsen}, \mathsf{low}\rangle$
$\langle \mathsf{PrivateNote}, \mathsf{high}\rangle$
$\langle \mathsf{MedicalRecord}, \mathsf{low}\rangle$

As before we also assume that the location MAC can only be modified by privileged users.

Firstly, we enforce the Bell-LaPadula security policy [Gol99] to illustrate that AspectKE can enforce a well-known mandatory access control policy. Then we will enforce our example policy, with small modifications based on the aspects that enforce Bell-LaPadula policy.

If we enforce the Bell-LaPadula security policy, the first part of the policy states that a subject is allowed to read or input data from any object provided that the subject's security level dominates that of the object. In our case, this guarantees *no read up*: that is, low subjects (nurses) cannot read high objects (private notes) but can only read low objects (medical records); however, high subjects (doctors) can access both kinds of records.

The *no read up* part of the policy can be enforced by aspects as follows:

$$\begin{aligned}&\mathsf{A}^{read}_{\mathsf{p1_{B1}}}[user :: \mathbf{read}(\_, recordtype, \_, \_, \_)@\mathsf{EHDB}]\\&\quad\triangleq \mathbf{case}(\neg(\mathbf{test}(user, \mathsf{low})@\mathsf{MAC} \wedge \mathbf{test}(recordtype, \mathsf{high})@\mathsf{MAC}))\\&\qquad\mathbf{proceed};\\&\quad\mathbf{break}\end{aligned}$$

The second part of the policy (a simplified form of Bell-LaPadula star property [Gol99]) states that a subject can write to any object provided that the security level of the object dominates that of the subject (*no write down*). In our case high subjects (doctors) cannot write low objects (medical records) but low subjects (nurses) can write to both kinds of records.

The *no write down* of the policy can be enforced by the aspect below:

$$\begin{aligned}&\mathsf{A}^{out}_{\mathsf{p1_{B2}}}[user :: \mathbf{out}(\_, recordtype, \_, \_, \_)@\mathsf{EHDB}]\\&\quad\triangleq \mathbf{case}(\neg(\mathbf{test}(user, \mathsf{high})@\mathsf{MAC} \wedge \mathbf{test}(recordtype, \mathsf{low})@\mathsf{MAC}))\\&\qquad\mathbf{proceed};\\&\quad\mathbf{break}\end{aligned}$$

Additionally, we have an aspect for the **read** action to prevent users from reading records without specifying the record type, and an aspect for the **in** action to prevent users from reading and deleting records:

$$\begin{aligned}&\mathsf{A}^{read}_{\mathsf{p1_{B3}}}[user :: \mathbf{read}(\_, !recordtype, \_, \_, \_)@\mathsf{EHDB}]\\&\quad\triangleq \mathbf{break}\\&\mathsf{A}^{in}_{\mathsf{p1_{B4}}}[user :: \mathbf{in}(\_, \_, \_, \_, \_)@\mathsf{EHDB}]\\&\quad\triangleq \mathbf{break}\end{aligned}$$

These aspects correctly enforce our policy about reading patient records. However, the no write down policy is not quite right for our example, instead we depart from the Bell-LaPadula policy and define:

$$\begin{aligned}&\mathsf{A}^{out}_{\mathsf{p1_{B2'}}}[user :: \mathbf{out}(\_, recordtype, \_, \_, \_)@\mathsf{EHDB}]\\&\quad\triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{high})@\mathsf{MAC})\\&\qquad\mathbf{proceed};\\&\quad\mathbf{break}\end{aligned}$$

This aspect allows doctors to write any kind of record.

The aspect $\mathsf{A}^{out}_{\mathsf{p1}_{\mathsf{B2}'}}$ together with $\mathsf{A}^{read}_{\mathsf{p1}_{\mathsf{B1}}}, \mathsf{A}^{read}_{\mathsf{p1}_{\mathsf{B3}}}, \mathsf{A}^{in}_{\mathsf{p1}_{\mathsf{B4}}}$ reflect a mandatory access control model which satisfies our policy. In this case we only allow high users (doctors) to write patient records. Hence nurse NsOlsen in Example 3.3 cannot execute the third action as it will be blocked by $\mathsf{A}^{out}_{\mathsf{p1}_{\mathsf{B2}'}}$, which however would be allowed with $\mathsf{A}^{out}_{\mathsf{p1}_{\mathsf{B2}}}$ from the Bell-LaPadula security policy. □

3.3.3 Role-Based Access Control

Role-based access control (RBAC) [SCFY96] is another access control mechanism which allows central administration of security policies and is often more flexible and elegant for modelling security policies. The simplest model in the RBAC family is $RBAC_0$, where there are three sets of entities called *user*, *role*, and *permission*. A user can be assigned multiple roles (role assignment) and a role can have multiple permissions (permission assignment) to corresponding operations. In addition, the user can initiate a *session* during which the user activates some subset of roles that he or she has been assigned. A user can execute an operation only if the user's active roles have the permission to perform that operation.

Example 3.5 To implement the security policy for patient records, we use a model that does not differentiate a user's assigned role and active role (we assume that the assigned roles of all users are activated by default), so we only need location RDB with tuples $\langle user, role \rangle$:

$\langle$DrSmith, Doctor$\rangle$
$\langle$NsOlsen, Nurse$\rangle$

For permission assignment we also need a location to describe each role's permission. This can be done by storing tuples $\langle role, object, capability \rangle$ at PDB:

$\langle$Doctor, MedicalRecord, read$\rangle$
$\langle$Doctor, PrivateNote, read$\rangle$
$\langle$Doctor, MedicalRecord, out$\rangle$
$\langle$Doctor, PrivateNote, out$\rangle$
$\langle$Nurse, MedicalRecord, read$\rangle$

Once more we assume that the locations RDB and PDB can only be modified by privileged users.

The following aspects then implement the required policy:

$$\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{p1C1}}[user :: \mathbf{read}(_, recordtype, _, _, _)@\mathsf{EHDB}]\\
&\quad \triangleq \mathbf{case}(\exists role \in \{\mathsf{Doctor}, \mathsf{Nurse}\} :\\
&\qquad\qquad (\mathbf{test}(user, role)@\mathsf{RDB} \wedge \mathbf{test}(role, recordtype, \mathsf{read})@\mathsf{PDB}))\\
&\qquad \mathbf{proceed};\\
&\qquad \mathbf{break}\\
&\mathsf{A}^{in}_{\mathsf{p1C2}}[user :: \mathbf{in}(_, recordtype, _, _, _)@\mathsf{EHDB}]\\
&\quad \triangleq \mathbf{case}(\exists role \in \{\mathsf{Doctor}, \mathsf{Nurse}\} :\\
&\qquad\qquad (\mathbf{test}(user, role)@\mathsf{RDB} \wedge \mathbf{test}(role, recordtype, \mathsf{in})@\mathsf{PDB}))\\
&\qquad \mathbf{proceed};\\
&\qquad \mathbf{break}\\
&\mathsf{A}^{out}_{\mathsf{p1C3}}[user :: \mathbf{out}(_, recordtype, _, _, _)@\mathsf{EHDB}]\\
&\quad \triangleq \mathbf{case}(\exists role \in \{\mathsf{Doctor}, \mathsf{Nurse}\} :\\
&\qquad\qquad (\mathbf{test}(user, role)@\mathsf{RDB} \wedge \mathbf{test}(role, recordtype, \mathsf{out})@\mathsf{PDB}))\\
&\qquad \mathbf{proceed};\\
&\qquad \mathbf{break}
\end{aligned}$$

These three aspects are useful for interrupting the execution when a user attempts to operate on EHR records with a concrete record type, which essentially relies on the tuples from RDB and PDB. They also show the benefit of admitting quantifiers into the conditional expressions.

Similar to the previous subsections, we have to enforce additional aspects for capturing user attempts to access EHR records without specifying the record type.

$$\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{p1C4}}[user :: \mathbf{read}(_, !recordtype, _, _, _)@\mathsf{EHDB}]\\
&\quad \triangleq \mathbf{break}\\
&\mathsf{A}^{in}_{\mathsf{p1C5}}[user :: \mathbf{in}(_, !recordtype, _, _, _)@\mathsf{EHDB}]\\
&\quad \triangleq \mathbf{break}
\end{aligned}$$

In the following sections we will use the role-based access control mechanism since it is best suited for enforcing security policies in a large organization. □

3.3.4 Advice for Retrofitting Policies

Now we will show how aspects in AspectKE can retrofit new security policies into an exiting system that is being developed/updated or has already been deployed. Concretely, when a new functionality has been introduced to the existing system we will show how we enforce new policies to cater for the new requirements posed by the add-on functionality (Section 3.3.4.1); when a policy has been proposed to refine existing policies, we will show how we enforce policies based on the existing functionality of the system (Section 3.3.4.2); on the other hand, sometimes additional functionality has to be introduced to the system to implement aspects which refine existing policies (Section 3.3.4.3).

Indeed, the possibility to refine, renew and retrofit security policies into an existing/evolving system is very important for IT systems. Taking the EHR system as an example, as the public debate about security standards (especially for privacy) evolves, governments have to modify the corresponding acts and guides. As a consequence, the security policy for an EHR system will undergo frequent change [BS04]. Moreover, the IT system itself will always be enhanced by new functionalities, which means that new policies need to be enforced. The National IT Strategy for the Danish Health Care Service states that "it is also important to acknowledge the fact that IT is not just implemented once and for all" [Nat03].

3.3.4.1 Enforcing Security Policy for New Functionality

AspectKE can be used for enforcing new security policies when a new functionality has been introduced at any phase of the system development. The running example in Section 2.2.3 introduces a functionality of the EHR system as regards to reading and writing from and to the EHR database (EHDB). Now we introduce another (new) function to the EHR system that enables a manager to add a patient, or delete information from the system.

In our programming model, each patient is represented by a location. A manager can add a new patient to the system as follows:

$$\mathsf{MgDavis} :: \quad \mathbf{newloc}(!patient).\mathbf{out}(patient, \mathsf{Patient})@\mathsf{RDB}$$

First a new location for the patient is created, then it is registered in the role database RDB by the **out** action. To delete a user one can simply perform an **in** action to the location RDB.

We shall now show how to enforce the following security policy regarding the manager role at the hospital [EB04], for this new-added functionality.

> *In the hospital, only managers are allowed to add a user to, or delete from the system.*

Example 3.6 The following aspects will enforce the above security policy.

$\mathsf{A}_{\mathsf{p2}}^{newloc}[user :: \mathbf{newloc}(\_)]$
$\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Manager})@\mathsf{RDB} \wedge \mathbf{test}(\mathsf{Manager}, \mathsf{Location}, \mathsf{newloc})@\mathsf{PDB})$
$\quad\quad \mathbf{proceed};$
$\quad\quad \mathbf{break}$
$\mathsf{A}_{\mathsf{p2}}^{out}[user :: \mathbf{out}(\_, \_)@\mathsf{RDB}]$
$\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Manager})@\mathsf{RDB} \wedge \mathbf{test}(\mathsf{Manager}, \mathsf{RDB}, \mathsf{out})@\mathsf{PDB})$
$\quad\quad \mathbf{proceed};$
$\quad\quad \mathbf{break}$
$\mathsf{A}_{\mathsf{p2}}^{in}[user :: \mathbf{in}(\_, \_)@\mathsf{RDB}]$
$\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Manager})@\mathsf{RDB} \wedge \mathbf{test}(\mathsf{Manager}, \mathsf{RDB}, \mathsf{in})@\mathsf{PDB})$
$\quad\quad \mathbf{proceed};$
$\quad\quad \mathbf{break}$

These aspects are composed in a way that is similar to the Example 3.5. Before using them we need to put the tuple $\langle \mathsf{MgDavis}, \mathsf{Manager} \rangle$ into RDB and the tuples $\langle \mathsf{Manager}, \mathsf{Location}, \mathsf{newloc} \rangle$, $\langle \mathsf{Manager}, \mathsf{RDB}, \mathsf{out} \rangle$, and $\langle \mathsf{Manager}, \mathsf{RDB}, \mathsf{in} \rangle$ into PDB. Finally, once again we assume that PDB can only be modified by privileged users. □

Example 3.6 shows that AspectKE can enforce a policy for new functionality (code), no matter when this functionality is developed, potentially in the entire life cycle of the EHR system. This is because the underlying aspect-oriented mechanism allows new aspects to intercept all join points (including a join point of new code) and give appropriate advice. Example 3.6 also shows how to give advice on an action (**newloc**) that creates new node in a net. Note that Examples 3.5 and 3.6 use the role-based access control model which shows that the tuples introduced above at PDB are good at expressing permissions that only directly rely on roles and objects. However, some permission assignments are more complex and therefore we shall also use logical formulae to express permission assignments in the following sections.

3.3.4.2 Refining Security Policy with Existing Functionality

AspectKE can be used to to refine an existing security policy at any phase of the system development. The following is a policy which can be considered as a refinement of the previous policy (enforced by Example 3.3-3.5) to protect patients' privacy. This policy is also inspired by [EB04]:

> *Private notes can only be viewed on the basis of the doctor-patient confidentiality – doctors cannot view private notes that were not created by themselves; nurses can only view recent medical records which were created after the patient has been admitted.*

Traditional programming paradigms normally necessitate modifying existing code to enforce this extra policy, while the aspect approach simply requires additional aspects.

Example 3.7 The first part of the policy can be expressed by the aspects shown below. These two aspects will prevent a doctor from reading a private note written by another doctor or reading a private note without clearly specifying the author of the note.

$$
\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{p3}_1}[user :: \mathbf{read}(\_, \mathsf{PrivateNote}, author, \_, \_)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Doctor})@\mathsf{RDB} \wedge (user = author)) \\
&\qquad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\neg\mathbf{test}(user, \mathsf{Doctor})@\mathsf{RDB}) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break} \\
&\mathsf{A}^{read}_{\mathsf{p3}_2}[user :: \mathbf{read}(\_, \mathsf{PrivateNote}, !author, \_, \_)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\neg\mathbf{test}(user, \mathsf{Doctor})@\mathsf{RDB}) \\
&\qquad \mathbf{proceed}; \\
&\qquad \mathbf{break}
\end{aligned}
$$

Note that the second case of $\mathsf{A}^{read}_{\mathsf{p3}_1}$ and $\mathsf{A}^{read}_{\mathsf{p3}_2}$ allow any users registered in RDB except doctors to read a private note, which includes users taking roles as nurses. Allowing nurses to read a private note is not problematic, as these two aspects only reflect the intention of this policy. Preventing nurses to read a private note is enforced in the previous policy.

These two aspects are supposed to work together with those aspects defined in the Examples 3.3 to 3.5. For example if a nurse tries to read a private note, these aspects will **proceed** the execution but aspects in Examples 3.3-3.5 will suggest **break**. Another example is if a doctor tries to read a private note written by other doctors, aspects in Examples 3.3-3.5 will allow the action to **proceed** whilst these aspects will **break** the execution. Since **break** takes precedence over **proceed**, the final decision suggested from this advice will be to block the execution in both cases.

The second part of the policy says that nurses can only read recent medical records which can be expressed by the following aspects where, once again, the aspects should be considered in conjunction with those of Examples 3.3 to 3.5. These two aspects will prevent a nurse from reading a past medical record or

reading a medical record without specifying the created time.

$$
\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{p3}_3}[user :: \mathbf{read}(\_, \mathsf{MedicalRecord}, \_, createdtime, \_)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB} \wedge createdtime = \mathsf{Recent}) \\
&\qquad \mathbf{proceed}; \\
&\quad \mathbf{case}(\neg\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB}) \\
&\qquad \mathbf{proceed}; \\
&\quad \mathbf{break} \\
&\mathsf{A}^{read}_{\mathsf{p3}_4}[user :: \mathbf{read}(\_, \mathsf{MedicalRecord}, \_, !createdtime, \_)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\neg\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB}) \\
&\qquad \mathbf{proceed}; \\
&\quad \mathbf{break}
\end{aligned}
$$

□

Note that no new functionality is required to enforce this policy, as these aspects only rely on the existing program (i.e., node RDB).

3.3.4.3 Refining Security Policy with New Functionality

Sometimes refining an existing security policy is necessary. We have to introduce additional functionality to support the implementation of the aspects for policy refinement. Now we consider the following security policy that restricts certain locations in the hospital from accessing to the database [Bez98]:

> *A nurse can only read medical records of the patients who are in the wards located on the nurse's working floor. Furthermore, the nurse can only access medical records through the computers that are located on that specific floor. But in the emergency room, a nurse does not have this restriction.*

Example 3.8 To express this policy as aspects we shall assume that the current location database CLDB records every user's current location information (indicating that they are using computers at that location), and that the assigned location database ALDB stores every user's assigned room (e.g. for nurses this is their working floors and rooms which they are responsible for; for patients this is the floors and wards that they are on). These two databases store tuples with the same fields $\langle user, floor, room\rangle$ and can only be modified by privileged users.

The set Floor contains the actual floors of the hospital (e.g. f1,f2). The set Room contains the actual rooms of the hospital and includes two types of rooms: ordinary wards (e.g. w1,w2) and the special room EmergencyRoom.

The appropriate advice can now be expressed as follows:

$$
\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{p4}_1}[user :: \mathbf{read}(patient, \mathsf{MedicalRecord}, \_, \_, \_)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB} \wedge \\
&\qquad\quad \exists floor \in \mathsf{Floor} : \mathbf{test}(user, floor, \mathsf{EmergencyRoom})@\mathsf{CLDB}) \\
&\qquad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB} \wedge \\
&\qquad\quad \exists floor \in \mathsf{Floor}\ \exists room \in \mathsf{Room} : (\ \mathbf{test}(user, floor, room)@\mathsf{CLDB} \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{test}(user, floor, room)@\mathsf{ALDB} \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{test}(patient, floor, room)@\mathsf{ALDB})) \\
&\qquad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\neg\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB}) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break} \\
&\mathsf{A}^{read}_{\mathsf{p4}_2}[user :: \mathbf{read}(!patient, \mathsf{MedicalRecord}, \_, \_, \_)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB} \wedge \\
&\qquad\quad \exists floor \in \mathsf{Floor} : \mathbf{test}(user, floor, \mathsf{EmergencyRoom})@\mathsf{CLDB}) \\
&\qquad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\neg\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB}) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break}
\end{aligned}
$$

In $\mathsf{A}^{read}_{\mathsf{p4}_1}$, the first case caters for the situation where a nurse is working in the emergency room; the second case allows the **read** action when a nurse is trying to access the medical record for a patient who is on the same ward/floor as where the nurse is currently at and assigned to work, the third case allows a user who is not a nurse to perform the **read** action. The aspect $\mathsf{A}^{read}_{\mathsf{p4}_2}$ is similar to the aspect $\mathsf{A}^{read}_{\mathsf{p4}_1}$ except that it does not contain the second case. This is because reading a medical record without specifying the name of the patient is not acceptable for a nurse who is not working in the emergency room.

As in Example 3.7, these aspects will work together with all previously introduced security polices. Moreover, these aspects are both in the spirit of role-based access control, and they demonstrate that when composing aspects in AspectKE for larger and more complex security policies of an organization, role-based access control can be very efficacious, as has already been observed in the literature [SCFY96]. □

Note new functionality has to be introduced to enforce this policy such as the newly introduced nodes for databases (e.g., CLDB, ALDB). This new functionality can be developed as part of the main logic of the EHR system, or can be merely developed and maintained to enforce security policies. We observe that although we might need to extend the functionality of the system's main logic to enforce certain security policies, the policies themselves are still described in aspects (even though they directly or indirectly rely on new nodes/processes), which are still separated from the main logic.

3.4 Concluding Remarks

In this chapter we presented basic features of AspectKE with formal semantics, where aspects can trap KLAIM actions before they are actually performed. We showed the usefulness of this language by enforcing various access control models, namely mandatory access control, discretionary access control and role based access control model, and by showing how aspects can retrofit policies into existing systems. The policies presented in this chapter are primarily extracted from a case study on building a secure electronic health care workflow system.

The language constructs presented here will be re-used in Chapter 4 when introducing the advanced features of AspectKE, as well as in Chapter 5 when we present AspectK.

Chapter 4

AspectKE: Trapping Processes

Classical reference monitors have difficulties of enforcing security policies based on the future behaviors of programs, rather they rely only on information gathered by monitoring execution steps [Sch00], and perform history-based dynamic checks. However, there are security policies that are concerned with information flow and thus cannot be implemented correctly without a security check of the overall behavior of the program.

For example, in a software system, remote evaluation involves the transmission of programs from a client to a server for subsequent execution at the server. However, as the programs transmitted might perform unintended operations at the server side, a security check is usually needed. A typical example of this is Java applets which can be transferred to a remote system and executed by the Java Virtual Machine (JVM). Since the unknown applets are not always written by trusted users, the JVM has certain mechanisms for ensuring that the applet will not be able to do malicious actions, e.g., the bytecode verifier [Ler01] and sandbox mechanism [Oak98].

As another example, in the EHR domain, rather than enforcing the primary use of data policies for direct patient care as shown in the last chapter, there is a trend to define and enforce secondary use of data policies. Here data is used outside of direct health care delivery that includes activities such as analysis, research, public health etc, even though it still lacks a robust infrastructure of

policies and is surrounded with complex ethical, political, technical and social issues [SBH+07]. Compared with primary use of data, whose focus is on regulating *"someone has some rights to access some data"*, it focuses on defining *"how the data can be used after it is released to someone"*. The enforcement of such policies requires security checks in the form of inspection of the flow of data.

In general, program analysis techniques concern computing reliable, approximate information about the dynamic behavior of programs [NNH05], and the derivation of useful information by simulating execution of all possible paths of the executing program. For example, type systems can be used to enforce various kinds of information flow as well as access control security policies (like Jif [MZZ+01] for Java).

In Section 4.1, we extend the AspectKE language presented in Chapter 3 so that AspectKE is enabled to not only trap the action, but also trap a process that is to be executed in the future. The process can be a process that is to be sent to a remote site, or a process continuation of a trapped action. This enables us to perform simple forms of program analyses, called *behavior analysis*, that syntactically inspect the trapped processes, i.e., actions in a new thread to be executed (at local/remote sites) or the remaining actions in the current thread. In the following Section 4.2, we will show how to use simple behavior analysis techniques to enforce various security policies that require checking of the future behavior of a program, the so-called *predictive access control policies*, and these techniques can detect and prevent execution of the potential malicious operations at the earliest stage.

4.1 Extended Syntax and Semantics

$cut \in \mathbf{Cut}$	cut	::=	$\cdots \mid \ell :: ca.X$
$ca \in \mathbf{CAct}$	ca	::=	$\cdots \mid \mathbf{eval}(X)@\ell$
$cond \in \mathbf{BExp}$	$cond$	::=	$\cdots \mid c \in set \mid \ell \in set \mid set = \emptyset$
$set \in \mathbf{Set}$	set	::=	$\cdots \mid \{c\} \mid \mathsf{Act}(X) \mid \mathsf{Loc}_c(X) \mid \mathsf{FV}(X) \mid \mathsf{FV}_c(X) \mid \mathsf{LC}(X)$ $\mid \mathsf{LC}_c(X) \mid \mathsf{LVar}_*$

Table 4.1: AspectKE Syntax - Aspects for Trapping Processes

Table 4.1 shows the extended syntax of AspectKE. The cut version of action (**CAct**) has been extended to include the **eval** action, whose parameter X is

$$\overrightarrow{asp} \vdash l_s :: (\mathbf{eval}(P')@l_0.P + \cdots) \rightarrow \begin{cases} \overrightarrow{asp} \vdash l_s :: P \mid\mid l_0 :: P' & \text{if } \Phi^{\overrightarrow{asp}}(l_s :: \mathbf{eval}(P')@l_0.P) \\ \overrightarrow{asp} \vdash l_s :: \mathbf{0} & \text{if } \neg\Phi^{\overrightarrow{asp}}(l_s :: \mathbf{eval}(P')@l_0.P) \end{cases}$$

Table 4.2: Reaction Semantics for action **eval** of AspectKE (on closed nets)

intended to bind its argument – the process P – in the ordinary action. Table 4.2 shows the semantics of the **eval** action. As was the case for the semantics of the other actions in Table 3.2, execution depends on the results of evaluation of the Φ function. When evaluated to **tt**, the **eval** action spawns a new process at the specified location before continuing with the following process; otherwise the action will be terminated and no process will be spawned.

The pointcut (**Cut**) in Table 3.1 has been extended with $\ell :: ca.X$, which not only binds the action, but also binds the program continuation after the trapped action to a variable X.

We shall modify the Well-formedness of cut defined in Section 3.1 to support the pointcut that captures the processes. First we slightly modify the *cl(cut)* function so that it takes a process parameter.

$$cl(l_s :: \mathbf{in}(!x, y)@l_0.X) = \langle l_s, x, y, l_0, X \rangle$$

Second we shall put a further restriction on the use of variables in cut. In Section 3.1, the use of a location variable u is not allowed in tests if $!u$ is used in a pointcut, here we additionally impose that in tests the use of a location variable shall not come from behavior analysis functions (that might return free variables), to avoid using free variables in tests.

As process variables are also included in *cl(cut)*, we need them to be pairwise distinct. For example, in an **eval** pointcut

$$cl(l_s :: \mathbf{eval}(Y)@l_0.X) = \langle l_s, Y, l_0, X \rangle$$

we require Y and X to be different, where Y binds a process to be evaluated as a new thread, while X binds a process to be executed after the **eval** action in the current thread.

The Φ function in Table 3.3 has to be replaced by that in Table 4.3, where it uses an updated version of the function *extract* and also records the process continuation:

$$extract(l :: \mathbf{out}(\ell_1, \cdots, \ell_n)@\ell'.P) = (l, \mathbf{out}, \ell_1, \cdots, \ell_n, \ell', P)$$

$$
\begin{array}{rcl}
\Phi^{A[cut]\triangleq body,\overrightarrow{asp}}(l :: a.P) & = & \text{case } check(\, extract(cut)\,;\, extract(l :: a.P)) \text{ of} \\
 & & \qquad \mathsf{fail}: \quad \Phi^{\overrightarrow{asp}}(l :: a.P) \\
 & & \qquad \theta: \quad (\mathbf{proceed} = [\![body\,\theta]\!]) \wedge \Phi^{\overrightarrow{asp}}(l :: a.P) \\
\Phi^{\varepsilon}(l :: a.P) & = & \mathbf{tt}
\end{array}
$$

Table 4.3: Trapping Aspects in AspectKE

The new version of *extract* will not directly effect the definition of *check* function, but we have to extend the *do* function defined in Table 3.4 to cope with process continuation P and variable X that are introduced by the *extract* function.

$$do(\, X\,;\, P\,) = [P/X]$$

Table 4.1 extends **BExp** and **set** in Table 3.1, which can be used for defining properties that require syntactic analysis of the processes to be executed (usually the continuation of the trapped action bound by X). The set-forming behavior analysis functions $\mathsf{Act}, \mathsf{Loc}_c, \mathsf{LC}, \mathsf{LC}_c, \mathsf{FV}, \mathsf{FV}_c$ will be explained in the following sections when needed, but we have collected their definitions in Table 4.4 where fv, bv, lc, cap and loc are the obvious extraction functions for free variables, bound variables, location constants, capabilities and target locations, respectively.

Note these functions expose different data-flow information of processes bound by process variables, and can be used to enforce *predictive access control policies*, namely access control policies that depend on the future behavior of a program.

Example 4.1 To illustrate how aspects in the extended AspectKE trap a KLAIM program and extract its properties, the following aspect gives advice to the running example in section 2.2.3.

$$
\begin{array}{l}
\mathsf{A}_2^{read}[user :: \mathbf{read}(\_,\_,\_,\_,\_)@\mathsf{DrSmith}.X] \\
\quad \triangleq \mathbf{case}(\mathbf{out} \in \mathsf{Act}(X)) \\
\qquad \mathbf{break}; \\
\qquad \mathbf{proceed}
\end{array}
$$

This aspect traps a **read** action of processes running at location DrSmith, when reading a tuple with five fields. The process continuation of the trapped action will be recorded in variable X. Function Act returns all actions in processes represented by X. If the actual process bound by X contains any **out** actions, the aspect will break the execution of the action and its continuation process. Otherwise, the action continues.

$\mathsf{Act}(P_1 \mid P_2)$	$=$	$\mathsf{Act}(P_1) \cup \mathsf{Act}(P_2)$
$\mathsf{Act}(\Sigma_i a_i.P_i)$	$=$	$\bigcup_i (\{cap(a_i)\} \cup \mathsf{Act}(P_i))$
$\mathsf{Act}(*P)$	$=$	$\mathsf{Act}(P)$
$\mathsf{Loc}_c(P_1 \mid P_2)$	$=$	$\mathsf{Loc}_c(P_1) \cup \mathsf{Loc}_c(P_2)$
$\mathsf{Loc}_c(\Sigma_i a_i.P_i)$	$=$	$\bigcup_i (\{loc(a_i) \mid cap(a_i) = c\} \cup \mathsf{Loc}_c(P_i))$
$\mathsf{Loc}_c(*P)$	$=$	$\mathsf{Loc}_c(P)$
$\mathsf{LC}(P_1 \mid P_2)$	$=$	$\mathsf{LC}(P_1) \cup \mathsf{LC}(P_2)$
$\mathsf{LC}(\Sigma_i a_i.P_i)$	$=$	$\bigcup_i (lc(a_i) \cup \mathsf{LC}(P_i))$
$\mathsf{LC}(*P)$	$=$	$\mathsf{LC}(P)$
$\mathsf{LC}_c(P_1 \mid P_2)$	$=$	$\mathsf{LC}_c(P_1) \cup \mathsf{LC}_c(P_2)$
$\mathsf{LC}_c(\Sigma_i a_i.P_i)$	$=$	$\bigcup_i (\{lc(a_i) \mid cap(a_i) = c\} \cup \mathsf{LC}(P_i))$
$\mathsf{LC}_c(*P)$	$=$	$\mathsf{LC}_c(P)$
$\mathsf{FV}(P_1 \mid P_2)$	$=$	$\mathsf{FV}(P_1) \cup \mathsf{FV}(P_2)$
$\mathsf{FV}(\Sigma_i a_i.P_i)$	$=$	$\bigcup_i (fv(a_i) \cup (\mathsf{FV}(P_i) \setminus bv(a_i)))$
$\mathsf{FV}(*P)$	$=$	$\mathsf{FV}(P)$
$\mathsf{FV}_c(P_1 \mid P_2)$	$=$	$\mathsf{FV}_c(P_1) \cup \mathsf{FV}_c(P_2)$
$\mathsf{FV}_c(\Sigma_i a_i.P_i)$	$=$	$\bigcup_i (\{fv(a_i) \mid cap(a_i) = c\} \cup (\mathsf{FV}_c(P_i) \setminus bv(a_i)))$
$\mathsf{FV}_c(*P)$	$=$	$\mathsf{FV}_c(P)$

Table 4.4: Behavior Analysis Functions

Suppose we have a system that contains the same net as in running example of Section 2.2.3 and aspect A_2^{read}:

$$\begin{array}{ll} & \mathsf{A}_2^{read}[user :: \mathbf{read}(\_,\_,\_,\_,\_)@\mathsf{DrSmith}.X] \\ \textbf{let} & \quad \triangleq \mathbf{case}(\mathbf{out} \in \mathsf{Act}(X)) \qquad \textbf{in} \\ & \qquad \mathbf{break}; \\ & \quad \mathbf{proceed} \end{array}$$

$$\begin{array}{ll} & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\ || & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\ || & \mathsf{DrSmith} :: \mathbf{read}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, !content)@\mathsf{EHDB}. \\ & \qquad \mathbf{out}(\mathsf{Alice}, content)@\mathsf{DrSmith}. \\ & \qquad \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB} \end{array}$$

and some steps of execution (omitting the aspect definition):

$$\begin{array}{ll} & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\ || & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\ || & \mathsf{DrSmith} :: \mathbf{read}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, !content)@\mathsf{EHDB}. \\ & \qquad \mathbf{out}(\mathsf{Alice}, content)@\mathsf{DrSmith}. \\ & \qquad \mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB} \\ \rightarrow & \\ & \mathsf{EHDB} :: \langle \mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrHansen}, \mathsf{Past}, \mathsf{alicetext} \rangle \\ || & \mathsf{EHDB} :: \langle \mathsf{Bob}, \mathsf{PrivateNote}, \mathsf{DrJensen}, \mathsf{Recent}, \mathsf{bobtext} \rangle \\ || & \mathsf{DrSmith} :: \mathbf{0} \end{array}$$

Aspect A_2^{read} traps the **read** action, whose result substitution is

$$[\mathsf{DrSmith}/user, \\ \mathbf{out}(\mathsf{Alice}, content)@\mathsf{DrSmith}.\mathbf{out}(\mathsf{Alice}, \mathsf{MedicalRecord}, \mathsf{DrSmith}, \mathsf{Recent}, \mathsf{newtext})@\mathsf{EHDB}/X]$$

Here $\mathsf{Act}(P) \subseteq \{\mathbf{out}, \mathbf{in}, \mathbf{read}, \mathbf{eval}, \mathbf{newloc}\}$ is the set of capabilities performed by the process P (see Table 4.4). In this case, Act returns set $\{\mathbf{out}\}$, the case condition evaluates to **tt**, thus the aspect breaks the execution of this action and its continuation process. □

4.2 Advice for Data Usage

In this section, we now show how to use the extended AspectKE to enforce security policies that require behavior analyses of processes to be executed in future, and we clarify the meaning of set-forming behavior analysis functions (from **Set** in Table 4.1 and 4.4) through examples – enforcing several predictive access control EHR security polices to the target EHR system presented in Section 2.2.3. In Section 4.2.1, we show how to enforce policies by utilizing the

set-forming functions that check properties of remotely evaluating processes. In Section 4.2.2, we show how to enforce policies by checking properties of the continuation process at the current thread.

4.2.1 Remote Evaluation

Using the action **eval**, AspectKE can easily express a process's remote evaluation. Moreover, using behavior analysis, AspectKE can check the content of the transmitted process by composing various aspects that embody different security considerations. This gives us a flexible way of controlling the use of remote processes. We will enforce a security policy in a distributed mobile environment that has to consider both direct and indirect access to tuple spaces, and in the latter case AspectKE shows great usefulness.

Consider a policy concerning removal of data from the system [EB04]:

> *Doctors, nurses and patients are not allowed to delete records from the database – only the administrator of the database can do that.*

Thus, in terms of AspectKE, only the administrator is allowed to perform an **in** action on the EHR database.

Example 4.2 In section 3.3.1-3.3.3, we introduced aspects for restricting the **in** actions to access the EHR database when enforcing a basic access control policy. Here we shall slightly update them to reflect the new policy. As we prefer to use the role-based access control model, we show how to update the relevant aspects in Section 3.3.3.

For aspect $\mathsf{A}^{read}_{\mathsf{p1c_1}}$, we need to add role Administrator to role set, while updating tuple spaces RDB with tuple $\langle \mathsf{AdWalker}, \mathsf{Administrator} \rangle$, and PDB with tuples $\langle \mathsf{Administrator}, \mathsf{MedicalRecord}, \mathsf{in} \rangle$ and $\langle \mathsf{Administrator}, \mathsf{PrivateNote}, \mathsf{in} \rangle$.

In aspect $\mathsf{A}^{in}_{\mathsf{p1c_5}}$, we have forbidden all users to delete records from the EHR database. Now we relax this requirement and replace it with the following aspect.

$$\begin{aligned}&\mathsf{A}^{in}_{\mathsf{p5}}[user :: \mathbf{in}(\_, !recordtype, \_, \_, \_)@\mathsf{EHDB}]\\ &\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Administrator})@\mathsf{RDB}\\ &\qquad \mathbf{proceed};\\ &\quad \mathbf{break}\end{aligned}$$

This breaks direct attempts to perform **in** actions, only actions by the administrator are allowed.

This advice only deals with direct attempts to delete data; we also have to cater

for processes like

$$\mathsf{NsOlsen} :: \quad \mathbf{eval}\Big(\mathbf{in}(patient, !recordtype, !author, !createdtime, !subject)@\mathsf{EHDB}\Big) \\ @\mathsf{AdWalker}$$

where anyone (e.g., NsOlsen) who spawns an arbitrary process on the administrator AdWalker's node can behave as an administrator and delete the records.

This behavior can be captured by an aspect for **eval** actions that targets the AdWalker location, without using any behavior analysis functions.

$$\begin{array}{l} \mathsf{A}^{eval}_{\mathsf{p5_A}}[user :: \mathbf{eval}(Y)@\mathsf{AdWalker}.X] \\ \quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Administrator})@\mathsf{RDB}) \\ \qquad \mathbf{proceed}; \\ \quad\ \mathbf{break} \end{array}$$

However, this aspect is too restrictive as it disallows the possibility for other users to perform well-behaved actions on behalf of an administrator (e.g., **out**,**read** etc.).

Using behavior analysis functions, we are able to check the process in advance so that less restrictive policies can be enforced. For example, the following aspect prevents remotely spawned **in** actions on AdWalker, but allows other types of action.

$$\begin{array}{l} \mathsf{A}^{eval}_{\mathsf{p5_B}}[user :: \mathbf{eval}(Y)@\mathsf{AdWalker}.X] \\ \quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Administrator})@\mathsf{RDB}) \\ \qquad \mathbf{proceed}; \\ \quad\ \mathbf{case}(\mathbf{test}(user, \_)@\mathsf{RDB} \wedge \neg(\mathbf{in} \in \mathsf{Act}(Y))) \\ \qquad \mathbf{proceed}; \\ \quad\ \mathbf{break} \end{array}$$

Here $\mathsf{Act}(P)$ is the set of capabilities performed by the process P (see Table 4.4). It follows that any **in** actions within Y will be trapped. There is no restriction for actions other than **in** actions, so remote code can still perform actions like **out** and **read**.

We may want to be more liberal and allow **in** actions on locations distinct from EHDB. To do so we introduce $\mathsf{Loc}_{\mathbf{in}}(P)$ to be the set of locations ℓ, where $\mathbf{in}(\cdots)@\ell$ occur in P; note that this set may include location constants as well as location variables (see Table 4.4).

Rather than aspects $\mathsf{A}^{eval}_{\mathsf{p5_A}}$ and $\mathsf{A}^{eval}_{\mathsf{p5_B}}$, we could use the aspect:

$$\begin{array}{l} \mathsf{A}^{eval}_{\mathsf{p5_C}}[user :: \mathbf{eval}(Y)@\mathsf{AdWalker}.X] \\ \quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Administrator})@\mathsf{RDB}) \\ \qquad \mathbf{proceed}; \\ \quad\ \mathbf{case}(\mathbf{test}(user, \_)@\mathsf{RDB} \wedge (\mathsf{LVar}_\star \cup \{\mathsf{EHDB}\}) \cap \mathsf{Loc}_{\mathbf{in}}(Y) = \emptyset) \\ \qquad \mathbf{proceed}; \\ \quad\ \mathbf{break} \end{array}$$

where $\mathsf{LVar}_\star$ is the set of all location variables in the program. This aspect allows users to perform an **in** action on behalf of an administrator when the target locations will never be EHDB, which is the least restrictive aspect for enforcing the same policy. □

Example 4.2 shows that whilst security policies may be very simple to enforce superficially, execution of remotely spawned code might easily invalidate policies which appear to be reasonable. Using aspects, we are able to elegantly update the enforcement of the security policy to cater for this. Furthermore, the examples also illustrate AspectKE's capability of checking the remote code before it is executed, which gives the users greater flexibility to enforce a less restrictive but more precise policy.

One might wonder whether combining the use of **newloc** and **eval** actions will break the above security policies. Consider the following example:

$$\begin{aligned}\mathsf{NsOlsen} :: \quad & \mathbf{newloc}(!u).\mathbf{out}(u, \mathsf{Administrator})@\mathsf{RDB}.\\ & \mathbf{eval}\Big(\mathbf{in}(patient, !recordtype, !author, !createdtime, !subject)@\mathsf{EHDB}\Big)@u\end{aligned}$$

Here NsOlsen tries to create a new location, register it to RDB, then execute the **in** action from the new location. This attempt will not work: if policy $\mathsf{A}_{\mathsf{p2}}^{newloc}$ and $\mathsf{A}_{\mathsf{p2}}^{out}$ from Example 3.6 are still enforced, NsOlsen is neither able to create any new location nor update RDB since only a Manager can do that.

In fact, all aspects defined in this paper will directly **break** any action executed at a location that has not been registered in RDB, and this includes all attempts to use **newloc** and **eval** to bypass the security policies as shown above, because only the manager can update RDB through aspects defined in Example 3.6.

4.2.2 Using Program Continuations

Now we show how AspectKE can trap the continuation process and use behavior analysis functions to get the future behavior of the executing process, which enables the advice to control and avoid executing the malicious processes as early as possible.

As we have mentioned before, our society is moving in the direction of trying to exploit patient healthcare records that already exist for new purposes (known as secondary use of data). At the Canadian Institute of Health Research [Can02] it is stated that "health research based on the secondary use of data contributes to our present level of understanding of the causes, patterns of expression and natural history of diseases." This raises new challenges for developing an effective system to ensure people's rights to privacy and confidentiality: decisions concerning access control decisions are not only based on the right of access of

different principals but should also examine how the data is to be used after access has been provided.

For example, researchers who are making secondary use of data should not be able to access the identity of patients. Therefore we want to prevent them from executing a process like

$$\mathsf{RsMiller} :: \quad \mathbf{read}(patient, !recordtype, !author, !createdtime, !subject)@\mathsf{EHDB}$$

which explicitly specifies the *patient* whose records the researcher is interested in reading. We can easily compose aspects that forbid the researcher from performing such actions (but allow nurses, doctors, etc ...) that are similar to those aspects that have been shown in Section 3.3.

In order for the researcher to blindly get a patient's healthcare record, the researcher may perform a process such as

$$\mathsf{RsMiller} :: \quad \mathbf{read}(!patient, recordtype, !author, !createdtime, !subject)@\mathsf{EHDB}$$

The difference between the **read** action in this program and those in the previous examples is the use of ! in front of *patient*, i.e., the researcher does not specify which patient's healthcare record is to be read. However, after the researcher has obtained the healthcare record, s/he can still *use* the patient's identity. We might want to prevent the researcher from executing a process like

$$\begin{aligned}\mathsf{RsMiller} :: \quad & \mathbf{read}(!patient, \mathsf{MedicalRecord}, !author, !createdtime, !subject)@\mathsf{EHDB}. \\ & \mathbf{out}(patient, subject)@\mathsf{Publication}\end{aligned} \tag{4.1}$$

whereas it would be acceptable to execute the process

$$\begin{aligned}\mathsf{RsMiller} :: \quad & \mathbf{read}(!patient, \mathsf{MedicalRecord}, !author, !createdtime, !subject)@\mathsf{EHDB}. \\ & \mathbf{out}(subject)@\mathsf{Publication}\end{aligned} \tag{4.2}$$

since the second program will not *use* the identity of the patient whose record has been selected.

The following policy is extensively discussed and accepted around the world and is specified directly or indirectly in a number of codes (e.g. [Dep03b, Can02, Dan00, SBH+07]):

> *Researchers should not read and use the patient's healthcare records of an EHR system in a way that might potentially expose the identity of the patient.*

Example 4.3 The following aspects, which replace the aspects $\mathsf{A}^{read}_{\mathsf{p1c_1}}$ and $\mathsf{A}^{read}_{\mathsf{p1c_4}}$ in Example 3.5, enforce this policy, which enforce both the basic access control policies for doctors and nurses, and policies for the researchers regarding their rights to read EHR records. It is necessary to revise the aspects because our previous development was only for primary use of data.

$$\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{рб}_1}[user :: \mathbf{read}(patient, recordtype, _, _, _)@\mathsf{EHDB}.X] \\
&\quad \triangleq \mathbf{case}(\exists role \in \{\mathsf{Doctor}, \mathsf{Nurse}\} : \\
&\qquad\qquad (\mathbf{test}(user, role)@\mathsf{RDB} \wedge \mathbf{test}(role, recordtype, \mathsf{read})@\mathsf{PDB})) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break} \\
&\mathsf{A}^{read}_{\mathsf{рб}_2}[user :: \mathbf{read}(patient, !recordtype, _, _, _)@\mathsf{EHDB}.X] \\
&\quad \triangleq \mathbf{break}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{рб}_3}[user :: \mathbf{read}(!patient, recordtype, _, _, _)@\mathsf{EHDB}.X] \\
&\quad \triangleq \mathbf{case}(\exists role \in \{\mathsf{Doctor}, \mathsf{Nurse}\} : \\
&\qquad\qquad (\mathbf{test}(user, role)@\mathsf{RDB} \wedge \mathbf{test}(role, recordtype, \mathsf{read})@\mathsf{PDB})) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\mathbf{test}(user, \mathsf{Researcher})@\mathsf{RDB} \wedge \neg(patient \in \mathsf{FV}(X))) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break} \\
&\mathsf{A}^{read}_{\mathsf{рб}_4}[user :: \mathbf{read}(!patient, !recordtype, _, _, _)@\mathsf{EHDB}.X] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Researcher})@\mathsf{RDB} \wedge \neg(patient \in \mathsf{FV}(X))) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break}
\end{aligned}$$

Aspect $\mathsf{A}^{read}_{\mathsf{p1}_{\mathsf{C1}}}$ is replaced by divided into aspects $\mathsf{A}^{read}_{\mathsf{рб}_1}$ and $\mathsf{A}^{read}_{\mathsf{рб}_3}$, according to whether a patient name is clearly specified or not. Similarly, $\mathsf{A}^{read}_{\mathsf{p1}_{\mathsf{C4}}}$ is replaced/divided by aspects $\mathsf{A}^{read}_{\mathsf{рб}_2}$ and $\mathsf{A}^{read}_{\mathsf{рб}_4}$. In both cases, additional conditions regarding researchers are only added to aspects when a patient name is not specified, i.e., $\mathsf{A}^{read}_{\mathsf{рб}_3}$ and $\mathsf{A}^{read}_{\mathsf{рб}_4}$. In these two aspects, the behavior analysis function FV is used, which returns the set of free variables of P (see Table 4.4).

In our example the aspect $\mathsf{A}^{read}_{\mathsf{рб}_3}$ will bind X with the **out** actions of the above two programs: in program (4.1) with $\mathbf{out}(patient, subject)@\mathsf{Publication}$, and in program (4.2) with $\mathbf{out}(subject)@\mathsf{Publication}$. Thus $\neg(patient \in \mathsf{FV}(X))$ would be evaluated to **ff** for the first case and to **tt** for the second case. Using this extra information from the behavior analysis function, the advice can be based on some properties of the future execution of the continuation process (in particular, whether or not *patient* will ever be used). And the suggestions from aspect $\mathsf{A}^{read}_{\mathsf{рб}_3}$ would be **break** for the first case and **proceed** for the second one. Note at the point that the access control decision has been made, $!patient$ in the join point **read** action is still a defining occurrence variable and thus does not bind with any actual location yet, and behavior analysis merely analyses the future behavior of process based on its static information. □

The above aspect is too restrictive since it forbids the execution of meaningful **read** actions as well. As one of the case studies performed by the Canadian Government [Can02] indicates:

In practice, the researchers might need to do some data linkage operations between different databases.

For example, we may allow the researcher to extract several records for the same patient from different databases and put that information together as in the process

$$\begin{aligned}\mathsf{RsMiller} :: \quad & \mathbf{read}(!patient, \mathsf{MedicalRecord}, !author, !createdtime, !subject1)@\mathsf{EHDB}.\\ & \mathbf{read}(patient, \mathsf{MedicalRecord}, !author, !createdtime, !subject2)@\mathsf{EHDB2}.\\ & \mathbf{out}(subject1, subject2)@\mathsf{Publication}\end{aligned} \tag{4.3}$$

where we introduce another EHR database EHDB2 that is located at another hospital.

Now there are two databases so we need to consider which policies are suitable for each of them, respectively. For illustration purposes, we might simply demand that the second database has the same security policy as the original one. Thus the second **read** action would be denied, since the original policy prohibits a researcher from reading a specific patient's healthcare record. Because of this, establishing data linkage in the direct manner of program (4.3) will never work.

To make the data linkage work we can restrict the researcher's access to the data through a trusted location (EHDB for example) by remote evaluation, and let the trusted location perform the actual data linkage actions. In this way the policy will allow the second **read** action whenever it is executed from the trusted location:

$$\begin{aligned}\mathsf{RsMiller} :: \quad & \mathbf{eval}\Big(\mathbf{read}(!patient, \mathsf{MedicalRecord}, !author, !createdtime, !subject1)@\mathsf{EHDB}.\\ & \quad \mathbf{read}(patient, \mathsf{MedicalRecord}, !author, !createdtime, !subject2)@\mathsf{EHDB2}.\\ & \quad \mathbf{out}(subject1, subject2)@\mathsf{Publication}\\ & \Big)@\mathsf{EHDB}\end{aligned} \tag{4.4}$$

Example 4.4 Revisiting the aspects $\mathsf{A}_{\mathsf{p6}_1}^{read}$–$\mathsf{A}_{\mathsf{p6}_4}^{read}$ for the original database in Example 4.3, we shall enable the trusted locations to read and perform data linkage actions from the EHR databases. For simplicity, we assume that these trusted locations are all EHR databases. Thus when a database performs data linkage actions, the aspects should not disallow the use of *patient* in the second **read** of program (4.4), but they should definitely prevent the **out** action with *patient* as parameter, where we put the same restriction for the researchers. We may replace the aspects $\mathsf{A}_{\mathsf{p6}_1}^{read}$–$\mathsf{A}_{\mathsf{p6}_4}^{read}$ with the following aspects for enforcing such a security policy for databases, as well as the policies for doctors, nurses,

researchers that we discussed in the previous examples:

$$
\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{p7}_1}[user :: \mathbf{read}(patient, recordtype, \_, \_, \_)@dest.X] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB} \wedge \exists role \in \{\mathsf{Doctor}, \mathsf{Nurse}\} : \\
&\qquad\qquad (\mathbf{test}(user, role)@\mathsf{RDB} \wedge \mathbf{test}(role, recordtype, \mathsf{read})@\mathsf{PDB})) \\
&\qquad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB} \wedge \\
&\qquad\qquad (\mathbf{test}(user, \mathsf{DataBase})@\mathsf{RDB} \wedge \mathbf{test}(\mathsf{DataBase}, recordtype, \mathsf{read})@\mathsf{PDB})) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\neg\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB}) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break} \\
&\mathsf{A}^{read}_{\mathsf{p7}_2}[user :: \mathbf{read}(patient, !recordtype, \_, \_, \_)@dest.X] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB} \wedge \\
&\qquad\qquad (\mathbf{test}(user, \mathsf{DataBase})@\mathsf{RDB} \wedge \mathbf{test}(\mathsf{DataBase}, recordtype, \mathsf{read})@\mathsf{PDB})) \\
&\qquad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\neg\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB}) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break} \\
&\mathsf{A}^{read}_{\mathsf{p7}_3}[user :: \mathbf{read}(!patient, recordtype, \_, \_, \_)@dest.X] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB} \wedge \exists role \in \{\mathsf{Doctor}, \mathsf{Nurse}\} : \\
&\qquad\qquad (\mathbf{test}(user, role)@\mathsf{RDB} \wedge \mathbf{test}(role, recordtype, \mathsf{read})@\mathsf{PDB})) \\
&\qquad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB} \wedge \\
&\qquad\qquad (\mathbf{test}(user, \mathsf{DataBase})@\mathsf{RDB} \vee \mathbf{test}(user, \mathsf{Researcher})@\mathsf{RDB}) \wedge \\
&\qquad\qquad \neg(patient \in \mathsf{FV}_{\mathbf{out}}(X))) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\neg\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB}) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break} \\
&\mathsf{A}^{read}_{\mathsf{p7}_4}[user :: \mathbf{read}(!patient, !recordtype, \_, \_, \_)@dest.X] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB} \wedge \\
&\qquad\qquad (\mathbf{test}(user, \mathsf{DataBase})@\mathsf{RDB} \vee \mathbf{test}(user, \mathsf{Researcher})@\mathsf{RDB}) \wedge \\
&\qquad\qquad \neg(patient \in \mathsf{FV}_{\mathbf{out}}(X))) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\neg\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB}) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break}
\end{aligned}
$$

In these aspects, DataBase represents all EHR databases in the system and includes EHDB and EHDB2. $\mathsf{A}^{read}_{\mathsf{p7}_1}$ and $\mathsf{A}^{read}_{\mathsf{p7}_2}$ replaces $\mathsf{A}^{read}_{\mathsf{p6}_1}$ and $\mathsf{A}^{read}_{\mathsf{p6}_2}$ by introducing additional cases to allow databases to read EHR records with a specified patient name. $\mathsf{A}^{read}_{\mathsf{p7}_3}$ and $\mathsf{A}^{read}_{\mathsf{p7}_4}$ replace $\mathsf{A}^{read}_{\mathsf{p6}_3}$ and $\mathsf{A}^{read}_{\mathsf{p6}_4}$ by ensuring both databases and researchers can blindly read an EHR record if they never output the obtained patient name to other locations. This is achieved by using $\mathsf{FV}_{\mathbf{out}}(P)$ behavior analysis functions, which returns the set of free variables occurring in **out** actions in P. Now roles Researcher and DataBase can both

blindly read EHR records as long as they do not output the patient's identity, but only the role DataBase can explicitly specify which patient to read. These aspects will prevent the execution of program 4.1 and 4.3 but program 4.2 and 4.4 will be allowed. □

As demonstrated in the last subsection, if remote evaluation is allowed, we need to pay closer attention to the overall security of the system. For example, in the event that the researcher attempts to get a doctor to link databases and output the private information as in

$$\begin{aligned}\mathsf{RsMiller} :: \quad & \mathbf{eval}\Big(\mathbf{read}(!\mathit{patient}, \mathsf{MedicalRecord}, !\mathit{author}, !\mathit{createdtime}, !\mathit{subject1})@\mathsf{EHDB}.\\ & \quad \mathbf{read}(\mathit{patient}, \mathsf{MedicalRecord}, !\mathit{author}, !\mathit{createdtime}, !\mathit{subject2})@\mathsf{EHDB2}.\\ & \quad \mathbf{out}(\mathit{patient}, \mathit{subject1}, \mathit{subject2})@\mathsf{Publication}\\ & \Big)@\mathsf{DrSmith}\end{aligned} \tag{4.5}$$

or in the event that the researcher attempts to obtain the records of a patient whose name is selected from his own tuple space either before evaluating a process at an EHR database or during the evaluation procedure.

$$\begin{aligned}\mathsf{RsMiller} :: \quad & \mathbf{read}(!\mathit{patient})@\mathsf{RsMiller}.\\ & \mathbf{eval}\Big(\mathbf{read}(\mathit{patient}, \mathsf{MedicalRecord}, !\mathit{author}, !\mathit{createdtime}, !\mathit{subject1})@\mathsf{EHDB}.\\ & \quad \mathbf{read}(\mathit{patient}, \mathsf{MedicalRecord}, !\mathit{author}, !\mathit{createdtime}, !\mathit{subject2})@\mathsf{EHDB2}.\\ & \quad \mathbf{out}(\mathit{patient}, \mathit{subject1}, \mathit{subject2})@\mathsf{Publication}\\ & \Big)@\mathsf{DrSmith}\end{aligned} \tag{4.6}$$

$$\begin{aligned}\mathsf{RsMiller} :: \quad & \mathbf{eval}\Big(\mathbf{read}(!\mathit{patient})@\mathsf{RsMiller}.\\ & \quad \mathbf{read}(\mathit{patient}, \mathsf{MedicalRecord}, !\mathit{author}, !\mathit{createdtime}, !\mathit{subject1})@\mathsf{EHDB}.\\ & \quad \mathbf{read}(\mathit{patient}, \mathsf{MedicalRecord}, !\mathit{author}, !\mathit{createdtime}, !\mathit{subject2})@\mathsf{EHDB2}.\\ & \quad \mathbf{out}(\mathit{patient}, \mathit{subject1}, \mathit{subject2})@\mathsf{Publication}\\ & \Big)@\mathsf{DrSmith}\end{aligned} \tag{4.7}$$

we need an aspect that inspects the actions performed by a researcher performing an **eval** action on all the EHR databases or locations other than the EHR databases.

Example 4.5 This motivates the aspect:

$$
\begin{aligned}
&\mathsf{A}_{\mathsf{p7}}^{eval}[user :: \mathbf{eval}(Y)@dest.X] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Researcher})@\mathsf{RDB} \wedge \mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB}) \wedge \\
&\qquad\qquad \forall x \in \mathsf{LC}_{\mathbf{read}}(Y) : \neg(\mathbf{test}(x, \mathsf{Patient})@\mathsf{RDB}) \wedge \\
&\qquad\qquad \forall y \in \mathsf{Loc}_{\mathbf{read}}(Y) : (\mathbf{test}(y, \mathsf{DataBase})@\mathsf{RDB})\ ) \\
&\qquad\ \mathbf{proceed}; \\
&\qquad \mathbf{case}(\mathbf{test}(user, \mathsf{Researcher})@\mathsf{RDB} \wedge \\
&\qquad\qquad (\neg\mathbf{test}(dest, \mathsf{DataBase})@\mathsf{RDB} \wedge \mathbf{test}(dest, \_)@\mathsf{RDB}) \wedge \\
&\qquad\qquad \forall x \in \mathsf{Loc}_{\mathbf{out}}(Y) : (\mathbf{test}(x, \mathsf{DataBase}) \vee x \in \{dest\})\ ) \\
&\qquad\ \mathbf{proceed}; \\
&\qquad \mathbf{case}(\neg\mathbf{test}(user, \mathsf{Researcher})@\mathsf{RDB} \wedge \mathbf{test}(user, \_)) \\
&\qquad\ \mathbf{proceed}; \\
&\qquad \mathbf{break}
\end{aligned}
$$

The first case of the aspect ensures that a researcher can directly evaluate processes in the EHR databases if s/he is not able to get the name of the patient whose record s/he is trying to obtain either before evaluation or during evaluation (by reading a patient name from a location other than EHR databases). Note that $\mathsf{LC}_{\mathbf{read}}(P)$ is a set of location constants in read actions of process P (defined in Table 4.4). The second case guarantees that when sending a process to other remote locations, the process only contains **out** actions that are performed on the EHR databases (trusted locations) or the tuple space associated with that remote location. □

In summary, the examples in this section show the versatility of AspectKE when dealing with remote evaluation and future execution paths of processes. We illustrated the usefulness of each behavior analysis function when enforcing access control policies that require the future behavior of process. When selecting the appropriate behavior analysis functions, it is possible to enforce less restrictive policies and avoid the execution of malicious processes as early as possible. More importantly, in a highly complex and privacy related computing environment, with policies that are changed frequently such as the EHR system, enforcing various access control and data usage policies through security aspects shows the potential of being a very flexible and elegant approach.

4.3 Concluding Remarks and Related Work

In this chapter we presented the advanced features of AspectKE. By trapping processes and using behavior analysis functions, AspectKE can enforce policies that require explicit information flow, and thus predictive access control and secondary use of data policies can be intuitively specified and enforced.

The EHR examples presented, in Chapter 3 and this chapter, were chosen to illustrate the different characteristics of AspectKE, but they were also chosen so as to constitute a complete set of access control policies. Here we conjecture that our aspects indeed enforce a complete set of access control policies, but in order to validate our conjecture we need to apply a formal validation method to it, which is left for future work.

As we have finished presenting AspectKE, we will now mention some related work that is relevant for the theme of our language design but from other perspectives; namely distribution with aspect-oriented programming and security policy languages.

4.3.1 Distribution with Aspect-Oriented Programming

Much work has been done regarding how to deploy and weave aspects for distributed systems: some work is relevant for language design of distributed AOP with explicit distribution [NCT04, NSV+06, TT06], other work explores the implementation of AOP middleware to support distributed AOP [PSD+04, LJ06, TJSJ08].

AspectKE can be consider as a special distributed AOP. It naturally follows the KLAIM programming model and uses remote pointcuts [NCT04] that identify join points in a program running on different locations. However, AspectKE does not aim at enhancing the flexibility of mechanisms to deploy, instantiate and execute distributed aspects, e.g., support advice execution over remote hosts, as AWED [NSV+06] and ReflexD [TT06] have achieved. Rather, it focuses on integrating analysis components for reasoning about the local or mobile processes to support advanced access control in a distributed setting. Compared with other AOP, AspectKE provides a well-defined security enforcement mechanism to tuple space systems that supports process mobility.

AO4BPEL [CM04] is an aspect extension of the process-oriented composition language BPEL, which was originally designed for composing Web Services. Work in [CSZ04, CH05] discusses different principles of using AOP to implement coordination systems (in AspectJ), but they are not related to security. Recently, AspectKB [HNN09] has been proposed. It uses Belnap Logic to handle conflicts when distributed advices are composed in a coordination environment.

4.3.2 Security Policy Languages

Binder [DeT02] and Cassandra [BS04] are very powerful logic-based security policy languages, and both are based on datalog logic-programming language.

In [BS04], there are substantial case studies performed using Cassandra based on the UK National Health Service procurement exercise. In AspectKE, our security policies are mainly expressed through logical formulae and predicates in the aspects. The big difference is that AspectKE provides predicates that can describe future events. There are other prominent policy languages like Protune [BO05], Rei [KFJ03], Ponder [DDLS00], and KeyNote [BFIK99], which can express basic access control policies very well. However, only Ponder and Rei can express usage control through obligation policies but, unlike our approach, neither language can enforce them and they have to trust that the party receiving the data uses it in proper ways [DHS07].

Chapter 5

AspectK: Generalization

In chapter 4, we primarily focused on enriching the *pointcut* and *advice condition* that can be expressed in aspect, building on top of basic features of AspectKE (Chapter 3). In this chapter, we shall extend the basic features of AspectKE from the *type of advice* point of view, so that aspect will not only **break** or **proceed** the process execution, but also introduce additional behavior around them similar to AspectJ's *before* and *after* advice. The language formulated is called AspectK. We will then introduce the *open joinpoints* that commonly exist in coordination languages, and show how they are addressed in AspectKE and AspectK. After that we discuss the formulation of other extensions and problems of supporting the *around* pointcut in the presence of open joinpoint.

5.1 Syntax and Semantics

5.1.1 Syntax

The AspectK's syntax is somewhat similar to that of AspectKE (that only traps actions) presented in Section 3.1. Here we focus on the differences between the two.

Compared to AspectKE's syntax shown in Table 3.1, we have extended and

$body \in \mathbf{Advice}$	$body$	::=	**case** $(cond)$ $sbody$; $body$ | $sbody$
	$sbody$	::=	as **break** | as **proceed** as
$as \in \mathbf{Act}^*$	as	::=	$a.as$ | ε
$ca \in \mathbf{CAct}$	ca	::=	$\cdots$ | **eval**$(X)@\ell$

Table 5.1: AspectK Syntax - More Type of Advice

replaced some of them, as is displayed in Table 5.1. Here, we introduce *as* (in $\mathbf{Act}^*$) that represents a sequence of actions, which can be inserted in front of **break** and **proceed**, or inserted after **proceed**. We also extends *ca* (in $\mathbf{CAct}^*$) that supports the cut version of the **eval** action.

Inserting actions in front of or after **break** and **proceed** gives us the possibility to offer similar functions like *before*, *after* or even *around* advice as in mainstream aspect-oriented programming languages like AspectJ [KHH+01]. There can exist problems for fully supporting the *around* advice, due to the *open join-points* in coordination languages such as KLAIM. We will discuss that later in this chapter.

In case multiple aspects trap one action, all the *before* actions are executed in declaration order, then the original action (in case of no **break**), and finally the *after* actions in reverse declaration order. As in AspectKE, the keyword **break** takes precedence over keyword **proceed**.

We have to slightly modify the well-formedness of cuts defined in Chapter 3, and additionally introduce the following requirement: when $!u$ is used in the cut pattern, u should only occur in the *after* actions, but not in the *before* action. This is because the aspects in AspectK (and AspectKE) intercept the execution of an action before it is actually performed. Thus $!u$ (defined in pointcuts of **read**, **in** or **newloc**) is not bound with a concrete value yet when executing the *before* advice. Otherwise it will cause free variables in *before* advice.

5.1.2 Semantics

The semantics is given by a one-step reduction relation on well-formed systems, nets and actions. The reaction rules are defined in Table 5.2, and we also re-use the structural congruence on nets defined in Table 2.2 and re-use the operation match in Table 2.4.

In Table 5.2, the reaction rules for systems and nets are straightforward. The

$$\frac{\overrightarrow{asp} \vdash N \to \overrightarrow{asp} \vdash N'}{\textbf{let } \overrightarrow{asp} \textbf{ in } N \to \textbf{let } \overrightarrow{asp} \textbf{ in } N'}$$

$$\frac{\overrightarrow{asp} \vdash N_1 \to \overrightarrow{asp} \vdash N_1'}{\overrightarrow{asp} \vdash N_1 \mid\mid N_2 \to \overrightarrow{asp} \vdash N_1' \mid\mid N_2} \qquad \frac{N \equiv M \quad \overrightarrow{asp} \vdash M \to \overrightarrow{asp} \vdash M' \quad M' \equiv N'}{\overrightarrow{asp} \vdash N \to \overrightarrow{asp} \vdash N'}$$

$$l_s :: (\underline{\mathbf{out}}(\overrightarrow{l})@l_0.P + \cdots) \to l_s :: P \mid\mid l_0 :: \langle \overrightarrow{l} \rangle$$

$$l_s :: (\underline{\mathbf{in}}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots) \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \to l_s :: P\theta \qquad \text{if } match(\overrightarrow{\ell^\lambda}; \overrightarrow{l}) = \theta$$

$$l_s :: (\underline{\mathbf{read}}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots) \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \to l_s :: P\theta \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \qquad \text{if } match(\overrightarrow{\ell^\lambda}; \overrightarrow{l}) = \theta$$

$$l_s :: (\underline{\mathbf{eval}}(P')@l_0.P + \cdots) \to l_s :: P \mid\mid l_0 :: P'$$

$$l_s :: (\underline{\mathbf{newloc}}(!u).P + \cdots) \to l_s :: P[l'/u] \mid\mid l' :: \mathbf{0} \qquad \text{with } l' \text{ fresh}$$

$$l_s :: (\underline{\mathbf{stop}}.P + \cdots) \to l_s :: \mathbf{0}$$

$$\frac{\overrightarrow{asp} \vdash l_s :: \Phi^{\overrightarrow{asp}}_{\mathbf{proceed}}(l_s :: \mathbf{out}(\overrightarrow{l})@l_0).P \to \overrightarrow{asp} \vdash N}{\overrightarrow{asp} \vdash l_s :: \mathbf{out}(\overrightarrow{l})@l_0.P + \cdots \to \overrightarrow{asp} \vdash N}$$

$$\frac{\overrightarrow{asp} \vdash l_s :: \Phi^{\overrightarrow{asp}}_{\mathbf{proceed}}(l_s :: \mathbf{in}(\overrightarrow{\ell^\lambda})@l_0).P \mid\mid N' \to \overrightarrow{asp} \vdash N}{\overrightarrow{asp} \vdash l_s :: \mathbf{in}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots \mid\mid N' \to \overrightarrow{asp} \vdash N}$$

$$\frac{\overrightarrow{asp} \vdash l_s :: \Phi^{\overrightarrow{asp}}_{\mathbf{proceed}}(l_s :: \mathbf{read}(\overrightarrow{\ell^\lambda})@l_0).P \mid\mid N' \to \overrightarrow{asp} \vdash N}{\overrightarrow{asp} \vdash l_s :: \mathbf{read}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots \mid\mid N' \to \overrightarrow{asp} \vdash N}$$

$$\frac{\overrightarrow{asp} \vdash l_s :: \Phi^{\overrightarrow{asp}}_{\mathbf{proceed}}(l_s :: \mathbf{eval}(P')@l_0).P \mid\mid N' \to \overrightarrow{asp} \vdash N}{\overrightarrow{asp} \vdash l_s :: \mathbf{eval}(P')@l_0.P + \cdots \mid\mid N' \to \overrightarrow{asp} \vdash N}$$

$$\frac{\overrightarrow{asp} \vdash l_s :: \Phi^{\overrightarrow{asp}}_{\mathbf{proceed}}(l_s :: \mathbf{newloc}(!u)@l_0).P \mid\mid N' \to \overrightarrow{asp} \vdash N}{\overrightarrow{asp} \vdash l_s :: \mathbf{newloc}(!u)@l_0.P + \cdots \mid\mid N' \to \overrightarrow{asp} \vdash N}$$

Table 5.2: Reaction Semantics of AspectK (on closed nets)

rules for the five actions come in pairs (note this is different from AspectKE). One rule takes care of the action when no advice is allowed to interrupt it; this is syntactically denoted by underlining.

The rules for non-underlined actions take the same shape and make use of function Φ defined in Table 5.3. Rather than return a boolean value as in AspectKE, the result of Φ is a sequence of actions trapping $l_s :: a$; as in AspectKE, $\overrightarrow{asp}$ is a global environment of aspects. The index f is either **proceed** or **break**. In general f will become **break** (changed in function κ, which is defined in Table 5.3) if at least one "**break**" advice applies, otherwise it will be **proceed**. In case of **proceed** the action $\underline{a}$ is eventually emitted, otherwise it will be dispensed with and be replaced with the **stop** action, killing all the subsequent actions. Recall that advice is searched in the order of declaration and applies in a parenthesis-like fashion.

$$
\begin{array}{rcl}
\Phi_f^{A[cut]\triangleq body,\overrightarrow{asp}}(l :: a) & = & \text{case } check(\, extract(cut)\,;\, extract(l :: a)) \text{ of} \\
 & & \qquad \mathsf{fail}: \quad \Phi_f^{\overrightarrow{asp}}(l :: a) \\
 & & \qquad \theta: \quad \kappa_f^{\overrightarrow{asp},l::a}(body\ \theta) \\
\Phi_f^{\varepsilon}(l :: a) & = & \text{case } f \text{ of} \\
 & & \qquad \mathbf{proceed}: \quad \underline{a} \\
 & & \qquad \mathbf{break}: \quad \underline{\mathbf{stop}}
\end{array}
$$

$$
\begin{array}{rcl}
\kappa_f^{\overrightarrow{asp},\ell::a}(\mathbf{case}\ cond\ sbody; body) & = & \text{case } [\![cond]\!] \text{ of} \\
 & & \qquad \mathbf{tt}: \quad \kappa_f^{\overrightarrow{asp},\ell::a}(sbody) \\
 & & \qquad \mathbf{ff}: \quad \kappa_f^{\overrightarrow{asp},\ell::a}(body) \\
\kappa_f^{\overrightarrow{asp},\ell::a}(sbody) & = & \text{case } sbody \text{ of} \\
 & & \qquad as_1\ \mathbf{proceed}\ as_2: \quad as_1.\Phi_f^{\overrightarrow{asp}}(l :: a).as_2 \\
 & & \qquad\quad as\ \mathbf{break}\ : \quad as.\Phi_{\mathbf{break}}^{\overrightarrow{asp}}(l :: a)
\end{array}
$$

Table 5.3: Trapping Aspects in AspectK

The function Φ re-uses function *check* (defined in Table 3.4) and function *extract* (defined in Section 3.2), which checks the applicability of each aspect in the aspect environment to the action being trapped, and produces the corresponding bindings of actual action parameters to the formal parameters of advice. If a cut does not match a normal action, fail is returned and the remaining aspects in the aspect environment will apply to the trapped action and search for further advices; Otherwise, we use κ (see Table 5.3) to recursively search for further advices, and starts searching advice from the current aspect.

The κ function processes the advice associated with a matching cut. The first clause in the definition processes conditional advices using the results of evalua-

tion of $[\![cond]\!]$. The function $[\![\ ]\!]$ evaluates its parameter (a boolean condition) in the expected way. The second clause deals with non-conditional advices which are either **proceed** or **break** advices. In the former case, the before actions and after actions sandwich a recursive call to Φ to find further applicable aspects. In the latter case, the before actions are performed and Φ is called recursively to find further applicable aspects taking care to record the fact that a **break** has been encountered (re-set Φ's f to **break**).

Note that we have to extend the *do* function defined in Table 3.4 to cope with process P for remote evaluation and variable X for **eval** action, which are introduced by the *extract* function.

$$do(\ X\ ;\ P\ ) = [P/X]$$

Eventually, when all aspects in the aspect environment have been considered, the second clause of Φ is invoked (see Table 5.3). If no **break** has been encountered, the underlined action is emitted, otherwise a **stop** is emitted. In the latter case, the program will terminate after all of the before actions have been executed.

5.2 Advice for Access Control with Logging

With the before and after actions, AspectK can express policy that requires logging actions. Here we will show an example based on the development of EHR system introduced in the previous chapters.

Let us loosen the restriction of the policy's second part introduced in Section 3.3.4.2, which is inspired by [EB04].

> *Normally Nurses can only view recent medical records, but they can view past medical records for special purposes. In the latter case, the system must supply some form of logging for accesses and access attempts. The manager should be made aware of repeated attempts by a nurse to access information beyond his/her rights.*

To express this policy, we assume there exists a logging database (LogDB) that keeps track of special activities by tuples

$$\langle user, action, patient, recordtype, createdtime \rangle$$

The tuple records a *user*'s *action*s performed on a patient, who has a specific *recordtype* created at *createdtime*.

In order to capture the repeated attempts by nurses to access information beyond their rights, we need nurses to consume special tokens before they perform

special activities. Each nurse has a limited number of tokens within a certain period, mainly maintained by managers, which is stored in location TokenDB with tuples $\langle user, token \rangle$. Each user has a number of copies of the tuple $\langle user, \mathsf{Token} \rangle$ in TokenDB, and they cannot be removed by other users (aspects for enforcing this are omitted). Once a nurse uses up his/her own tokens he/she can still perform the action, however, the activity will be sent to a location ManagerAlert so that managers will be notified by checking this location.

The aspect $\mathsf{A}_{\mathsf{p3}_3}$ defined in Example 3.7 in Section 3.3.4.2 forbids any nurses' attempts to read a past medical record. Here we replace it with aspects $\mathsf{A}^{read}_{\mathsf{p8}_1}$ and $\mathsf{A}^{read}_{\mathsf{p8}_2}$ to reflect the new policy. The only difference of these two new aspects are whether they capture an action when the user reads a specific patient's medical record (with parameter *patient* in pointcut), or just reads a medical record without specifying patient name (with parameter !*patient* in pointcut).

$$
\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{p8}_1}[user :: \mathbf{read}(patient, \mathsf{MedicalRecord}, \_, createdtime, \_)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB} \wedge createdtime = \mathsf{Recent}) \\
&\qquad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB} \wedge createdtime = \mathsf{Past} \wedge \\
&\qquad\qquad \mathbf{test}(user, \mathsf{Token})@\mathsf{TokenDB}) \\
&\qquad\quad \mathbf{in}(user, \mathsf{Token})@\mathsf{TokenDB} \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad\quad \mathbf{out}(user, \mathsf{read}, patient, \mathsf{MedicalRecord}, createdtime)@\mathsf{LogDB} \\
&\qquad \mathbf{case}(\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB} \wedge createdtime = \mathsf{Past} \wedge \\
&\qquad\qquad \neg\mathbf{test}(user, \mathsf{Token})@\mathsf{TokenDB}) \\
&\qquad\quad \mathbf{out}(user)@\mathsf{ManagerAlert} \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad\quad \mathbf{out}(user, \mathsf{read}, patient, \mathsf{MedicalRecord}, createdtime)@\mathsf{LogDB} \\
&\qquad \mathbf{case}(\neg\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB}) \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad \mathbf{break}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{A}^{read}_{\mathsf{p8}_2}[user :: \mathbf{read}(!patient, \mathsf{MedicalRecord}, \_, createdtime, \_)@\mathsf{EHDB}] \\
&\quad \triangleq \mathbf{case}(\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB} \wedge createdtime = \mathsf{Recent}) \\
&\qquad \mathbf{proceed}; \\
&\qquad \mathbf{case}(\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB} \wedge createdtime = \mathsf{Past} \wedge \\
&\qquad\qquad \mathbf{test}(user, \mathsf{Token})@\mathsf{TokenDB}) \\
&\qquad\quad \mathbf{in}(user, \mathsf{Token})@\mathsf{TokenDB} \\
&\qquad\quad \mathbf{proceed}; \\
&\qquad\quad \mathbf{out}(user, \mathsf{read}, patient, \mathsf{MedicalRecord}, createdtime)@\mathsf{LogDB} \\
&\qquad \mathbf{case}(\mathbf{test}(user, \mathsf{Nurse})@\mathsf{RDB} \wedge createdtime = \mathsf{Past} \wedge \\
&\qquad\qquad \neg\mathbf{test}(user, \mathsf{Token})@\mathsf{TokenDB}) \\
&\qquad\quad \mathbf{out}(user)@\mathsf{ManagerAlert} \\
&\qquad\quad \mathbf{proceed};
\end{aligned}
$$

out($user$, read, $patient$, MedicalRecord, $createdtime$)@LogDB
case(¬**test**($user$, Nurse)@RDB)
proceed;
break

Aspect $\mathsf{A}^{read}_{\mathsf{p8}_1}$ (and $\mathsf{A}^{read}_{\mathsf{p8}_2}$) extends aspects $\mathsf{A}_{\mathsf{p3}_3}$ for two additional cases of nurses reading past medical records. The first case will be executed if users still have their tokens left in the token database. In this case, before performing the **read** action, one token has to be removed by the **in** action; after performing the **read** action, the details of this activity will be sent to logging database. The second case will be executed if a user does not have any tokens left for the token databases. Instead of removing a token before performing the action, the name of the user will be sent to a location where managers can be made aware of what happened, and managers can thus check the logging database to see whether the nurses performed too many abnormal activities.

5.3 Open Joinpoints and Other Language Extensions

5.3.1 Open Joinpoints

Trapping an input (**in** or **read** in AspectKE and AspectK) action *before* a concrete tuple has been selected for input requires to cope with joinpoints that contain constructs for *binding* new variables (the notation $!u$ is used in AspectKE and AspectK) – we call these *open joinpoints*. This is considerably more challenging than the closed joinpoints of traditional aspect-oriented language like AspectJ [KHH+01]. To be more concrete, when we trap a method call in AspectJ, we trap the actual call, i.e. the method name with its actual parameters, rather than the definition of the method, i.e. the method name with its *formal* parameters; in other words AspectJ traps closed joinpoints rather than *open* joinpoints.

In this dissertation, both AspectKE and AspectK provide mechanisms (integrated in their semantics) to handle open joinpoints, which exist in their common base language KLAIM. Because of the open joinpoints, we have put additional restrictions in order to correctly use our languages. These are integrated into the well-formedness conditions for these process calculi presented before. Otherwise free variables would occur in advice. In AspectK, the variables (in pointcut) that bind new variables (in a open joinpoint) can only be used in actions *after* the **break** or **proceed**. In AspectKE, the variables (in a pointcut) that bind

new variables (in a open joinpoint) can only be used in conditions that are combined with behavior analysis functions, for checking data flow among variables in joinpoints.

More generally, coordination languages are distinguished from object oriented languages and web service languages [CM04] by their need to deal with open joinpoints. Similar considerations would apply if we were to incorporate aspects into process calculi that, like the π-calculus, allow a notion of open input (or input from the environment) but would not be necessary for calculi without this feature [And01, BJJR04, JJR03, WZL03, Wan01]. This calls for considerable care in designing a notion of advice where input actions are trapped *before* a concrete tuple is selected for input.

5.3.2 Discussion About Other Extensions

We have shown how to extend the model of AspectKE (that traps actions) so that the two types of basic advice, **break** and **proceed**, can be equipped with actions before or after the advice (to obtain some of the benefits of *around* advice). Thus, we can perform common tasks such as logging which other AOP languages normally can do. We illustrate the usefulness of the design through an EHR example.

There are different views as to whether the actions generated by an advice should be subject to further advices. Throughout the development we have taken the view that this is indeed desirable. But it is straightforward to modify Table 5.3 to use underlined *before* and *after* actions to accommodate the alternative view.

We now discuss other possibilities of extending the current design of AspectK.

5.3.2.1 Types of Advice

We did consider the incorporation of an *ignore* advice, as is commonly expressible in AOP languages by the *around* advice, but we found this to be a challenging extension.

To illustrate the problems, consider the following advice

$$\mathsf{A}_{\mathsf{IGNORE}}[u :: \mathbf{read}(!v)@l_{priv}] \triangleq \mathbf{ignore}$$

for simply ignoring inputs from a private location l_{priv}. The problem with this definition is that it might trap a **read** action occurring in the following process $l :: \mathbf{read}(!w)@l_{priv}.\mathbf{out}(w)@l_{print}$, which would then become $l : \mathbf{out}(w)@l_{print}$ and contains a free variable; however, our semantics does not ascribe meaning

to such processes. AspectJ does not have this problem, as it does not deal with open joinpoints.

Even a somewhat more useful advice

$$\mathsf{A}_{\mathsf{REDIRECT}}[u :: \mathbf{read}(!v)@l_{priv}] \triangleq \mathbf{ignore}\ u :: \mathbf{read}(!v)@l_{sandbox}$$

for redirecting inputs from a private location l_{priv} to a sandbox $l_{sandbox}$ is problematic. Once again consider the program $l :: \mathbf{read}(!w)@l_{priv}.\mathbf{out}(w)@l_{print}$ that is intended to become $l :: \mathbf{read}(!w)@l_{sandbox}.\mathbf{out}(w)@l_{print}$. The problem is that our current notion of substitution does not achieve this effect: while we can bind v to w to obtain the substitution $[w/v]$, we would not normally let the substitution change the defining occurrence $!v$ in $u :: \mathbf{read}(!v)@l_{sandbox}$ to $!w$ so as to yield the desired $u :: \mathbf{read}(!w)@l_{sandbox}$.

This can be solved by suitable extensions of our approach; in particular we can introduce special variables, e.g. β, that can be substituted also in defining occurrences and write

$$\mathsf{A}_{\mathsf{REDIRECT}}[u :: \mathbf{read}(!\beta)@l_{priv}] \triangleq \mathbf{ignore}\ u :: \mathbf{read}(!\beta)@l_{sandbox}$$

Then the program $l :: \mathbf{read}(!w)@l_{priv}.\mathbf{out}(w)@l_{print}$ would be correctly transformed to $l :: \mathbf{read}(!w)@l_{sandbox}.\mathbf{out}(w)@l_{print}$.

5.3.2.2 Local or Global Advice

For simplicity we have taken an approach where all advices are given in advance and are global in scope. It would be worthwhile if we can introduce new advice while limit the scope of its applicability. Indeed, it might be natural to consider the aspect environment as distributed and associated with locations. In that case it is appropriate to define another version of **newloc**, with a $\mathbf{newloc}(!u : \Gamma)$ construct with inference rule,:

$$l ::^{\overrightarrow{asp}} \mathbf{newloc}(!u : \overrightarrow{asp}').P \to l ::^{\overrightarrow{asp}} P[l'/u] \mid\mid l' ::^{\overrightarrow{asp}'} 0 \quad \text{with } l' \text{ fresh}$$

This would constitute a static treatment of scoped advice unlike the dynamic treatment in CaesarJ [AGMO06].

This would be useful when dealing with **eval** actions, defined with local advice. Then we might write an advice for executing a process in a sandbox as follows:

$$\begin{array}{l}\mathsf{A}_{\mathsf{SANDBOX}}[u ::^{\overrightarrow{asp}} \mathbf{eval}(X)@l_{sensitive}] \triangleq \\ \quad \mathbf{newloc}(!u_{sandbox} : \overrightarrow{asp}[\mathsf{A}_{\mathsf{BOXREAD}}[u ::^{\overrightarrow{asp}'} \mathbf{out}(v)@w] \triangleq u ::^{\overrightarrow{asp}'} \mathbf{out}(v)@u_{sandbox}]) \\ \quad \mathbf{ignore} \\ \quad u ::^{\overrightarrow{asp}} \mathbf{eval}(X)@u_{sandbox}\end{array}$$

When executing a program $l ::^{\overrightarrow{asp}} \mathbf{eval}(P)@l_{sensitive}.P'$ the advice transforms it to a process that evaluates the process P in a confined location and redirects all outputs to a confined location.

Clearly a number of additional extensions can be contemplated. For example we might want to have more powerful pointcut languages that allow patterns to bind over a number of parameters (in order to avoid having separate advice for each arity of the operations) or gives priorities to advice.

5.3.2.3 Trapping Processes

One may wonder why not apply the same ideas developed in Chapter 4 to AspectK: trap a process (either the continuation process or a process to be sent remotely), and then use behavior analysis functions that directly analyze a process.

The reason is that if we allow before and after actions around **break** or **proceed**, a safe behavior analysis would be very difficult to achieve, since the processes trapped might execute more actions (inserted by aspects at runtime) than planned. This is an interesting direction for future work and will require more powerful program analyses than the behavior analyses presented in Chapter 4.

5.4 Concluding Remarks

In this chapter we presented AspectK, which is built on top of the basic AspectKE features introduced in Chapter 3. AspectK allows actions *before* and *after* **proceed** or **break**, which can benefit crosscutting activities such as logging, as the *before*, after, or even *around* advices do in other aspect-oriented programming languages. However, it is not trivial to add advice for ignoring or redirecting given actions due to the *open joinpoints*, which commonly exists in coordination languages. On the contrary, the advanced features of AspectKE introduced in Chapter 4 provide various behavior analysis functions, and enable us to reason about the future execution of processes. This improves the capability of standard reference monitors which normally can only cope with history-based security policies. However, AspectKE does not allow actions before and after **proceed** or **break** as AspectK does. We argued it would be more challenging to support both extensions presented in Chapters 4 and 5, because a safe behavior analysis would require more advanced program analyses, and thus be difficult to develop. Yet we believe it would be interesting future work.

Chapter 6

AspectKE*: Programming Language

We have presented the AspectKE programming model in previous chapters and have shown it is useful to enforce a variety of security policies. Now we will bring the model into practice by building a programming language and runtime system based on it.

However, it is not practical to straightforwardly implement AspectKE's original semantics, because it models computation as rewriting process descriptions. As rewriting substitutes variables, AspectKE can predict future behavior of a process from the current process description without tracking data flow in the past. However, this also means performing program analysis for each step of program execution, which causes huge runtime overhead. In other words, the behavior analysis functions are defined with the assumption that analyses must access the runtime state of every variable in continuation processes, thus a *dynamic* syntax-based program analysis (behavior analysis) is needed to evaluate each analysis function at runtime, which is not an appropriate model for checking the future behavior of processes in reality.

Regarding how to check the behavior of processes in practice, static analysis is a candidate technique as it could check a program before execution [Nec97,LY99]. For example, a runtime system of Java checks binary code before execution. This approach, however, has two limitations. First, it lacks flexibility - once the

analyses are developed, it is difficult to reuse them: the programmers cannot easily access the analysis results and (re)define security policies, because those analyses and security policies are normally integrated with a compiler and a runtime system of the language. The second is expressiveness: static analyses are sometimes too restrictive to accurately enforce security policies in practice, due to the fact that they have to approximate properties of a program. On the contrary, as another common technique to check process's behavior, runtime monitoring is precise, yet comes at the price of execution time overhead and lacks the mechanism to look into future events. Our implementation relies on both static analysis and runtime monitoring approaches.

In this and the following chapters, we will present the design and practical implementation of the AspectKE* programming language, an AOP language based on a distributed tuple space system under the Java environment, which provides expressive language constructs to enforce basic as well as predictive access control policies. It is a proof-of-concept language that demonstrates practical and expressive language primitives for AspectKE along with an efficient implementation model. Our approach utilizes static analysis techniques to implement the language, but the language in return also effectively overcomes the limitations of static analysis raised above. Some of the main features of AspectKE* are as follows.

- It provides high-level program analysis predicates and functions that can be used as pointcuts in aspects, which enable programmers to easily express conditions based on future behavior of processes.
- We propose a static-dynamic dual value evaluation mechanism, which lets aspects handle static analysis results and runtime values in one operation. It enables programmers to specify policies' static and dynamic conditions in a uniform manner.
- We propose an efficient implementation strategy that gathers static information for program analysis predicates and functions before execution, and at runtime performs merely look up operations, which minimizes the runtime overheads caused by program analysis predicates and functions.

In this chapter, we present a proof-of-concept programming language AspectKE* that is based on AspectKE presented in Chapter 3 and Chapter 4. We present the design (in Section 6.1) and usage of AspectKE* by developing a relative low-cost and much simpler system (compared with the more demanding and complex secure EHR workflow system which is partly illustrated in previous chapters), - a secure distributed chat system, as another application demonstrating the usefulness of the AspectKE programming model and AspectKE* programming language (in Section 6.2). We will focus on presenting the language features and

discussing its uniqueness when compared with other aspect-oriented programming languages (in Section 6.3).

In the following chapters, we shall present the technical challenges of implementing AspectKE's runtime system, and evaluate the language.

6.1 The AspectKE* Programming Language

AspectKE* programming language provides much more user-friendly and practical language constructs to program AspectKE model. First we present its syntax with brief explanations. The syntax is listed in Table 6.1 (explained in Section 6.1.1) and Table 6.2 (explained in Section 6.1.2).

6.1.1 Project, Location, Node and Process

Now we shall introduce the language constructs listed in Table 6.1 for base programs.

A *program* is the top-level programming concept in AspectKE*, which represents an individual network with all the computing components. It consists of a project declaration, several location declarations and a few entity declarations.

A *project declaration* (in project_decl) defines the name space of this program. All entities (in entity_decl_list) within a program are defined under the same name space: compiling the same entity declaration under a different project declaration will generate different runtime representations of the entity; An aspect will be enforced to nodes and processes only if this aspect is under the same name space.

A *location declaration* (in location_decl) defines pseudo (logical) identities for nodes in a program. Each pseudo identity associates with a unique physical ip address (and port number) when the program is instantiated at runtime.

An *id* (in id) is a string that consists of numbers, alphabetic characters, underlines etc. It normally represents the identity of a specific language construct. When in a format surrounded with quotation marks such as "abc", it represents a constant in type string with value abc.

A *node declaration* (in node_decl) consists of tuple declarations and a process call. A tuple starts with the keyword data, and a process call starts with keyword process followed by the name of the process and a list of parameters. Note that parallel processes can be instantiated at a node if they are directly called in a

process invoked by the node.

A *process declaration* (in process_decl) starts with keyword proc. Its formal parameter list contains two types of variables: location and string. The body of the process declaration is block, which starts with a list of sequential statements and followed by a block of statements to be executed in parallel (in par_block). A *sequential statement* (in stmt) can be a local variable declaration in a block, or an action of type out, in, read, eval and newloc, whose semantics are the same as those defined in AspectKE, which outputs a tuple, inputs a tuple (read and delete a tuple), reads a tuple, instantiates a process, and creates a new node. *Parallel statements* (in par_stmt) in a parallel block will be instantiated at the same time, which can be a process declared in the project, an anonymous process with merely one action, or an anonymous process that has a body block.

Example 6.1 List 6.1 shows a Hello World program that demonstrates the basic usage of nodes and processes. In the program, a process at node Loc1 reads "hello" and "world" from its own tuple space and create a process at node Loc2 that outputs these words in a different order.

Line 1 declares the project name, Line 3 declares the logical locations in the program. Lines 5-13 define initial states of node Loc1 and Loc2. Node Loc1 consists of three tuples and one process. Node Loc2 is empty. Lines 15-20 define a process p1. Line 16 declares two local variables, which are bound to values by an input action. For example, the tuple ⟨baz,"word1",foo⟩ at Line 17 matches the tuple ⟨Loc2,"word1","hello"⟩ at node Loc1, and binds baz with Loc2, foo with "hello", respectively. Line 18 performs an in action, which reads a tuple ⟨Loc2,"word2","world"⟩ from Loc2 in a similar manner to read actions, but then removes the read tuple. Line 19 creates a process p2 with parameters "hello", "world" and Loc2 at node Loc2. The process p2 then executes two out actions that outputs "hello" and "world" onto the node "Loc2" in a different order.

After a successful execution of the program, node Loc1 contains tuples ⟨Loc2, "word1", "hello"⟩ and ⟨Loc1, "word1", "hello"⟩; while node Loc2 shall contain tuples ⟨"hello", "world"⟩ and ⟨"world", "hello"⟩.

□

6.1.2 Aspects

Now we shall introduce the language constructs listed in Table 6.2 for aspects.

An aspect (in aspect_decl) is formed by a name, a pointcut and a body that contains set declarations and an advice. In an aspect, all variables declared in the pointcut and set declarations can be used in the advice.

program	::=	project_decl location_decl_list entity_decl_list
entity_decl_list	::=	ε | node_decl | process_decl | aspect_decl
	|	entity_decl_list entity_decl_list
project_decl	::=	**project** id **;**
location_decl_list	::=	ε | location_decl | location_decl_list **;** location_decl_list
location_decl	::=	**location** id_list
id_list	::=	id | id_list **,** id_list
id	::=	*string*
node_decl	::=	**node** id **{** tuple_decl_list process_call_init **}**
tuple_decl_list	::=	ε | tuple_decl | tuple_decl **;** tuple_decl
tuple_decl	::=	**data** (id_list)
process_call_init	::=	ε | process_call **;**
process_call	::=	**process** id (actual_param_list)
actual_param_list	::=	ε | id | actual_param_list **,** actual_param_list
process_decl	::=	**proc** id (formal_param_list) block
formal_param_list	::=	ε | type id | formal_param_list **,** formal_param_list
type	::=	**location** | **string**
block	::=	**{** stmt par_block **}**
stmt	::=	ε | variable_decl | action | stmt **;** stmt
variable_decl	::=	type id_list
action	::=	**out** (id_list)**@**id | **in** (id_list)**@**id | **read** (id_list)**@**id
	|	**eval** (process_call)**@**id | **newloc**(id)
par_block	::=	ε | **parallel** **{** par_stmt **}**
par_stmt	::=	ε | process_call **;** | action **;** | block | par_stmt par_stmt

Table 6.1: AspectKE* Syntax - 1

aspect_decl	::=	**aspect** id **{ advice :** pointcut **{** set_decl_list advice **} }**
pointcut	::=	cut_action on target continuation
cut_action	::=	**out** (cut_param_list) | **in** (cut_param_list)
	|	**read** (cut_param_list) | **eval** (**process** id) | **newloc**(**location**)
cut_param_list	::=	ε | cut_param | cut_param_list **,** cut_param_list
cut_param	::=	id | **bound** type id | **unbound** type id | type
on	::=	ε | **&& on** (**bound location** id) | **&& on** (id)
target	::=	ε | **&& target** (**bound location** id) | **&& target** (id)
continuation	::=	ε | **&& continuation** (**process** id)
set_decl_list	::=	ε | **set** id = { set_el_list } | set_decl_list **;** set_decl_list
set_el_list	::=	ε | set_el | set_el_list **,** set_el_list
set_el	::=	id | action_name | **LVAR**
action_name	::=	**OUT** | **IN** | **READ** | **EVAL** | **NEWLOC**
advice	::=	case_stmt | if_stmt
case_stmt	::=	case_list default
case_list	::=	ε | **case** bool_expr suggestion | case_list case_list
default	::=	**default** suggestion
if_stmt	::=	**if** bool_expr suggestion **else** else_stmt
else_stmt	::=	if_stmt | suggestion
suggestion	::=	**proceed ;** | **terminate ;**
bool_expr	::=	bool_expr || bool_expr | bool_expr **&&** bool_expr
	|	**!** bool_expr | (bool_expr) | id == id | id **!=** id
	|	**element_of** (set_el , set) | **empty** (set)
	|	**test** (test_param_list)**@**id | **beused** (id, action_set, id)
	|	**beusedsafe** (id, action_set, set, id)
	|	**exist** (id, set) < bool_expr> | **forall** (id, set) <bool_expr>
test_param_list	::=	ε | test_param | test_param_list **,** test_param_list
test_param	::=	id | type
set	::=	id | **{** set_el_list **}**| **performed** (id) | **targeted** (action_set, id)
	|	**union** (set, set) | **intersection** (set, set)
action_set	::=	**{** action_list **}**
action_list	::=	ε | action_name | action_list**,** action_list

Table 6.2: AspectKE* Syntax - 2

```
1  project helloworld;
2
3  location Loc1,Loc2;
4
5  node Loc1{
6          data (Loc2,"word1","hello");
7          data (Loc2,"word2","world");
8          data (Loc1,"word3","hi");
9          process p1(Loc2);
10 }
11
12 node Loc2{
13 }
14
15 proc p1(location baz){
16         string foo, bar;
17         read(baz, "word1",foo)@Loc1;
18         in(baz,"word2",bar)@Loc1;
19         eval(process p2(foo,bar,baz))@baz;
20 }
21
22 proc p2(string foo, string bar, location baz){
23         out(foo, bar)@Loc2;
24         out(bar,foo)@baz;
25 }
```

Listing 6.1: Hello World Main Program

6.1.2.1 Pointcut

A *pointcut* (in pointcut) must clarify which join points it tries to capture, which is specified by conjuncting four *pointcut predicates*, namely cut_action, on, target and continuation. Note that all actions are potential join points in AspectKE*. The current implementation allows predicates in the pointcut to be connected by the && operator but not || nor !.

A pointcut matches a join point if all parameters in the first three predicates match the corresponding fields in a join point (the fourth predicate is not involved in the matching procedure). These parameters have two purposes: specify the matching rule of a join point, or expose the information of the actual parameters of a selected join point. Note that the information that can be associated with actual parameters is both dynamic and the pre-computed static data. The exact meaning of how the information is collected and used will be thoroughly discussed in the following part of the dissertation.

When capturing an out, in or read action, a cut action uses a *parameter list* (in cut_param_list) to match each field of a tuple. Only when the parameter list has the same length as the tuple, and all parameters matches their corresponding fields, can we say the parameter list matches a tuple. A *parameter* (in cut_param) can be a constant (id), which matches against a constant with the same value, a variable (with modifier bound or unbound), which matches against any constants by a bound variable and any variables by a unbound variable, or a don't-care pattern (indicated by only specifying a type), which matches against any constants and variables. In any case, the parameter will only match a field with a same type. When the parameter is a variable (bound or unbound) and is matched, it will bind static as well as runtime information of the matching field, and this information can be used in the advice. We shall clarify this later.

When capturing an eval action, a cut action uses a process variable to directly bind the process to be evaluated in the join point action. When capturing a newloc action, a cut action matches a newloc action no matter what location will be generated.

The optional on and target predicates match and/or bind the locations a join point action is performing at and performing to, respectively. The matching rule is the same as a parameter matching a tuple's field. They can be jointly used with cut_action.

The optional continuation predicate will not influence the matching result. It will simply bind a continuation process of a matched join point action.

6.1.2.2 Advice

It is possible to pre-define several sets in advice. The *element of a set* (in set_el) can be a constant, an action name or LVAR (a special reserved keyword for variables).

In the *advice* (in advice), either if-else statement (allowing "else if") or case statement with terminate or proceed in the branches can be written. It allows only one advice declaration per aspect and there is only one kind of advice. It uses *boolean expression* (in bool_expr) to express conditions and uses *advice suggestion* (in suggestion) to describe an advice's intervention to the matched join point: proceed will continue the execution of the join point while terminate will prohibit the execution of the current join point and its continuation processes.

The boolean expression is defined using classical boolean operators and boolean predicates such as: element_of (test whether a element belongs to a set), empty (test whether a set is empty), exist (test whether there is any element from a set satisfies a boolean condition) and forall (test whether all elements from a set

Predicate & Function	The Return Value
performed(z)	the set of potential actions that process z will perform.
targeted(acts,z)	the set of destination locations that the actions in set acts of process z will target to.
beused(v,acts,z)	true if variable v will be potentially used in any actions in set acts of process z.
beusedsafe(v,acts, locs,z)	true if any potential action acts in process z that uses variable v is targeted only to locations in locs.

Table 6.3: Program Analysis Predicates and Functions

satisfy a boolean condition). The test predicate is used for testing whether a specified tuple currently exists in the tuple space of a checked location (similar to read action, but does not allow unbound variables defined in a parameter list). We shall discuss predicates beused and beusedsafe shortly.

The *set* (in set) is used in the boolean expression, which can be a name of a predefined set, an anonymous set, and union or intersection of two sets. We shall discuss set expressions performed and targeted shortly.

6.1.2.3 Program Analysis Predicates and Functions

We introduce language constructs called *program analysis predicates and functions* that predict future behavior of a program, and therefore are useful for enforcing predictive access controls that refer future events of a program.

Table 6.3 summarizes the predicates and functions, which allow for checking different properties of the future behavior of a continuation process; i.e., the rest of the execution from the current join point, or a process to be evaluated locally or remotely. In the table, z is the continuation process of the captured action. acts is a collection of action names such as IN and OUT. v is a variable (it shall be declared in the pointcut). locs is a collection of locations. When computing a predicate/function on process z, the results are collected from process z and all processes spawned by z. In Chapter 7, we explain the implementation of those predicates and functions by using static analysis.

Example 6.2 To illustrate the basic usage of aspects, we will show a simple

aspect for the Hello World program introduced in Example 6.1. This aspect will terminate the execution of an eval action targeted at node Loc2 if the process to be executed remotely contains any out actions.

```
1 aspect helloworld_aspect{
2    advice: eval(process y) &&
3       on(bound location s) && target(Loc2){
4       if(element_of(OUT, performed(y)))
5          terminate;
6       else
7          proceed;
8    }
9 }
```

Listing 6.2: Hello World Aspect

In this aspect, the pointcut matches the join point action at Line 19 in Listing 6.1. This is because when the eval action is performed at Line 19, variable baz has a value of Loc2, which matches the parameter of target predicate. Predicates eval and on do not put further restrictions on join point matching. When matched, they will bind variables with the actual parameters correspondingly, in this case the variable s will bind Loc1, while process variable y will bind process p2.

The advice is an if statement, and uses the program analysis function performed, which returns the set of actions that might be performed in process p2. In our case, this set will contain out actions (at Lines 23 and 24 of process p2 in Listing 6.1). Thus the element_of predicate is true and the aspect suggests to terminate the execution of eval action. □

6.2 A Secure Distributed Chat Application

Enforcing security policies to a distributed system is challenging, especially when trusted components of a system have to work with untrusted ones. For example, while a user of a chat system trusts the programs running at the service provider's computers, he or she may need to run a chat client program developed by an untrusted third-party. In such a case, we need to ensure that the untrusted program does not perform any malicious operations.

In this section, we show how AspectKE* can be used to enforce security policies for a distributed chat application that contains untrusted components.

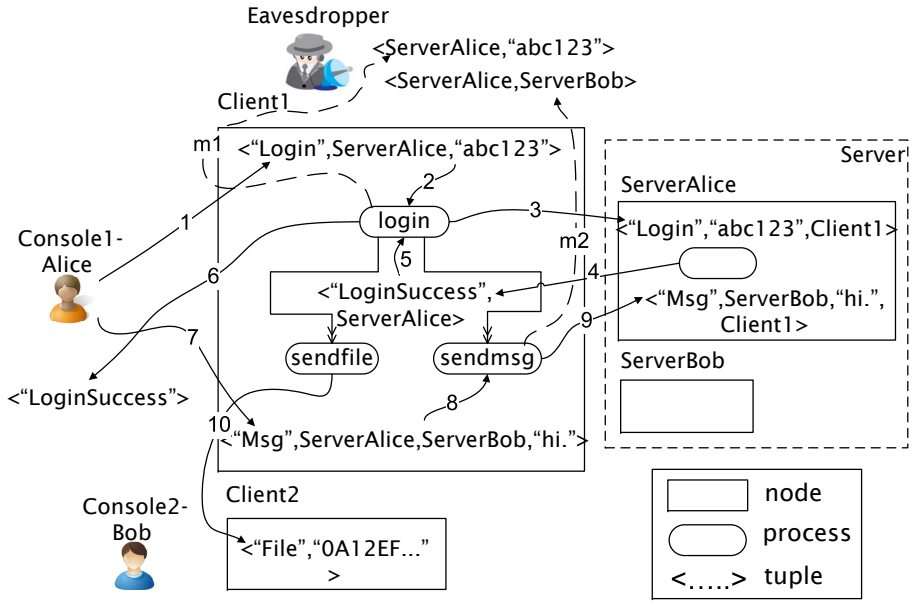

Figure 6.1: Overview of a Simplified Chat System

6.2.1 Background: Distributed Chat System and Security Policies

6.2.1.1 Distributed Chat System

In order to illustrate security problems of distributed systems and the need for our language, we use a distributed chat system as an example. Figure 6.1 shows an overview of the system, which consists of a server computer and a couple of users' client computers. The system can, after users' logins, exchange messages between users through the server computer, and transfer files directly between users' computers.

In the system, the users (i.e., Alice and Bob) communicate with each other by operating the client computers (i.e., Client1 and Client2) through console devices. Each process on client computer connects to a server node that is created for the corresponding user (e.g., ServerAlice) on the server computer. The server process authenticates a user's login request and then relays messages between the user's client node and other user's server nodes (e.g., ServerBob).

In the figure, a login procedure takes 6 steps, indicated by the arrows with number 1-6. (1) Alice makes a login request from Console1, which is observed by

Client1 as creation of a tuple of string "Login", the node of her server program (i.e., ServerAlice) and the password string that she typed in. (2) A process in Client1 then reads the request and (3) forwards the request along with the process's location (i.e., Client1) to ServerAlice. (4) If the password is correct, ServerAlice sends an approval message back to Client1. (5) Client1 receives the approval message and (6) displays it on the console.

After a successful login, the login process continues to spawn several processes to handle requests from this user and other users, including a process for message sending, as shown at steps 7-9. (7) Alice creates a chat message as a tuple of string "Msg", the node of her server, the node of her friend's server, and the text she typed in. (8) The process for sending messages will read this request and (9) deliver the chat message along with the process's location to her server (which will forward it to the friend's server). Another process is for transferring files, which (10) eventually sends a file directly to a friend's client program after negotiating with the server processes. (This part is omitted in the dissertation.)

Besides these intentional steps, the figure also illustrates two malicious operations that might be embedded in the client processes, namely, (m1) leak of the user's password. (m2) leak of the friendship between users.

6.2.1.2 Security Policies for the Chat System

We assume the following trust model for the chat system: the programs running on the server (namely ServerAlice and ServerBob) are trusted, while the programs running on Client1 and Client2 cannot be trusted, because the chat clients used are developed by a third-party. (Note that the consoles represent hardware devices, which shall be trusted.) The challenge is, how do we ensure the untrusted client programs cannot perform malicious operations.

First, we will set up security policies for the client programs, in order to clearly express what kinds of operations are considered as "malicious". Among many possible policies, we focus on the following three access control security policies. The first one is simple, the other rest two are predictive access control policies based on future behavior of a program.

Policy 1: When a client program sends a message tagged with "Msg" to a server program, it must correctly identify the sender information.

At step 9, the client creates a tuple ⟨"Msg",ServerBob,"hi.", Client1⟩, whose fourth field must be the sender. This policy prevents processes at an other client nodes forging a chat message from Alice.

Policy 2: A process in a client node is allowed to receive a message tagged with "Msg" from the console, if it will not send further messages to any node other than this user's server (that was assigned at the login procedure prior to the message).

For example, when Client1 receives a chat message sent from Alice to Bob (step 8), the continuation process can only output to Alice's server node (ServerAlice). This policy prevents the malicious process to receive chat messages from which it can compose malicious messages that will leak a pair of sender and receiver information (step m2) to an eavesdropper.

Policy 3: A process in a client node is allowed to receive a message tagged with "Login" from the console, if the client program will keep secrecy of the passwords in the message. Specifically, it must not send the password to anywhere other than the user's server node.

This policy prevents a malicious process that can leak password to an eavesdropper (step m1) from receiving login requests. Compared with Policy 2, this policy differs in that it concerns the messages that contain the password, while Policy 2 concerns any message. This is because some of the client process should be allowed to send messages to nodes besides the user's server node, for example, to send a file directly to another client node (step 10).

6.2.2 Distributed Chat System in AspectKE*

Let us see a part of the implementation of the chat system in AspectKE*. Listing 6.3 shows the node definition for Client1, one process will be instantiated when creating this node. Listing 6.4 shows a process definition within node Client1 (or Client2) that handles user login requests. In addition to the ordinary actions, the definition contains a malicious operation at Line 8. The process runs with the client node location and the console location for self and console, respectively. Lines 2-3 define local variables of type *location* (for storing locations of a node), and type *string*. Line 5 is a in action that reads and removes a tuple. The action waits for a tuple in the client node (as specified by self), which consists of three values: string "Login", any location, and any string. When such a tuple is created, the action deletes it, assigns the second and third elements in the tuple to userserver and password, and continues the subsequent statements afterwards. For example, when Alice makes a login request, a tuple ⟨"Login",ServerAlice,"abc123"⟩ is created in Client1. Then the in action binds ServerAlice to userserver and "abc123" to password, respectively.

Line 6 creates a tuple in a node by an out action. It creates, for example, a tuple ⟨"Login","abc123", Client1⟩ in the ServerAlice node. Similarly, Lines 8, 10 and 11 correspond to step m1, 5 and 6 in Figure 6.1. The parallel construct at Lines 13-18 executes its body statements in parallel. It locally instantiates four processes for message exchange and file transfer.

```
1  node Client1{
2    process clientlogin(Client1,Console1);
3  }
```

Listing 6.3: Node Client1

```
1  proc clientlogin(location self,location console){
2     location userserver;
3     string password;
4
5     in("Login",userserver,password)@self;
6     out("Login",password,self)@userserver;
7
8     out(userserver,password)@Eavesdropper;
9
10    in("LoginSuccess",userserver)@self;
11    out("LoginSuccess")@console;
12
13    parallel{
14       clientsendmsg(self,userserver,console);
15       clientreceivemsg(self,userserver,console);
16      clientsendfile(self,userserver,console);
17      clientreceivefile(self,userserver,console);
18    }
19 }
```

Listing 6.4: Process clientlogin

Note that the program listed above is malicious due to Line 8, which leaks password information to an eavesdropper.

Now let us take a look at the process clientsendmsg in Listing 6.5, which also contains a malicious operation.

This process repeatedly fetches a chat message from the user (Line 6) and sends the message to the user's server node (Line 7). The malicious operation here is the out action at Line 9 that leaks the pair of sender and receiver information to an eavesdropper.

```
1  proc clientsendmsg(location self,location userserver,
2       location console){
3     location friendserver;
4     string text;
5
6     in("Msg",userserver,friendserver,text)@self;
7     out("Msg",friendserver,text,self)@userserver;
8
9     out(userserver,friendserver)@Eavesdropper;
10
11    eval(process clientsendmsg(self,userserver,console))@self;
12 }
```

Listing 6.5: Process clientsendmsg

6.2.3 Security Aspects for Distributed Chat System

In this section, we will demonstrate how AspectKE* enforces the above three policies. In particular, we will show how program analysis predicates and functions can be used to enforce predictive access control policies (Policy 2 and 3).

6.2.3.1 Ensuring Correct Origin (Policy 1)

We will first show an aspect that enforces Policy 1, a basic access control policy. This policy requires that any out action of "Msg" message to a server node, like the one at Line 7 in Listing 6.5, should give the process's own location as the fourth element of the message.

Listing 6.6 defines the aspect that enforces this policy. It comprises its name ensure_origin, a pointcut (Lines 2-3) and advice body (Lines 4-8).

```
1  aspect ensure_origin{
2     advice: out("Msg",location,string,bound location client)
3        &&on(bound location s)&&target(bound location uid){
4        if(element_of(uid,{ServerAlice,ServerBob})&&s!=client)
5           terminate;
6        else
7           proceed;
8     }
9  }
```

Listing 6.6: Aspect for Ensuring the Correct Origin

Lines 2-3 define a pointcut that captures an out action targeted any node, where the out action can be performed by any process on any node. The parameters of the out predicate specify the values in the tuple that is to be sent by the out action, namely that the first element is "Msg", the second is any location, the third is any string, and the fourth is any location. Predicates on and target take the value of the captured process's location and destination of the action respectively. When it matches, the values of the process location, target location, and the values in the tuple to be sent, are bound to the variables in the pointcut. When a client process on Client1 executes out("Msg",ServerBob,"Hello",Client1)@ServerAlice, Client1, Client1 and ServerAlice are bound to variables client, s and uid, respectively.

Lines 4-8 are the body of the advice that terminates a process if the target location of the out action (uid) is either ServerAlice or ServerBob, and the fourth element of the tuple (client) is not the location on which the process is running (s). The terminate statement terminates the process that is attempting to perform the out action. Otherwise, the advice performs proceed statement that continues the process.

6.2.3.2 Protecting Chat Information (Policy 2)

Listing 6.7 shows an aspect that enforces Policy 2 by exploiting the program analysis functions.

```
1  aspect protect_message{
2     advice: in("Msg",bound location uid,location,string)&&
3        on(bound location s)&&target(bound location client)
4        &&continuation(process z){
5        if(element_of(client,{Client1,Client2})&&
6           !forall(x,targeted({OUT},z))<x==uid>)
7           terminate;
8        else
9           proceed;
10    }
11 }
```

Listing 6.7: Aspect for Protecting Chat Information

In the aspect, the pointcut at Line 2 captures the in action in clientsendmsg (Line 6 of Listing 6.5). The in action receives a request of sending a text message. When the pointcut matches, values ServerAlice, Client1 and Client1 are bound to variables uid, s and client respectively. Note that the predicate continuation captures the rest of the process, which is bound to variable z.

The condition at Lines 5 and 6 checks whether the action reads from a client

node, and the continuation process only sends messages to the user's server node, which is specified by the second element in the tuple.

The condition at Line 6 demonstrates the use of a program analysis function. First, the function targeted({OUT},z) returns all the destinations of out actions in process z. In the example, the destinations are userserver and Eavesdropper. Then the expression forall(x,...)<x==uid> checks if all the destination locations are the user's server node (uid).

The expression also demonstrates our *dual value evaluation mechanism* for uniformly performing static and dynamic checking. Since the advice runs at an in action, the destinations of future out actions might not be computed when the advice is being executed. In AspectKE*, the expression x==uid holds true either when the destination x of a future out action is predicted to have the same value as the one that is captured as uid, or when a future out action has a constant target location, which happens to be the same one in uid. Therefore, when advice captures the following action:

```
in("Msg", userserver, friendsserver, text)@self
```

where the value of userserver is ServerAlice, the expression x==uid holds for the destination of the following future action:

```
out("Msg", friendsserver, text, self)@userserver
```

because x and uid capture variables that have data flow between them (in fact, they are the same variable in this case). The expression x==uid does not hold for the future action:

```
out(userserver,friendsserver)@Eavesdropper
```

because x is a constant location Eavesdropper, which is different from the runtime value in uid, namely ServerAlice. This hybrid interpretation is useful when a client program replaces the malicious code – Line 9 in Listing 6.5– with the following operation, to backup messages for Alice [1]:

```
if(userserver==ServerAlice){
  out("MsgBackup",console,self,friendserver,text)
       @ServerAlice;
}
```

The aspect in Listing 6.7 will suggest *proceed* at Line 6 in Listing 6.5 when Alice executes the modified client program, however, it will *terminates* the in action when users other than Alice executes this client program. The is due to

[1]We use the if construct for readability, which has to be encoded into tuple operations in our current implementation of AspectKE*

the different comparison results between the runtime value bound to uid, and the unique static value (ServerAlice) collected from the out action by program analysis function targeted.

6.2.3.3 Protecting Passwords (Policy 3)

Listing 6.8 demonstrates the use of another program analysis predicate beusedsafe. This aspect enforces Policy 3. Compared with Policy 2, this policy terminate a process if particular data, but not the whole tuple, is potentially output to an untrusted place. It requires more complicated analyses that can simultaneously track the flow of variables and the flow of potential target locations. In AspectKE*, this can be achieved simply by using the beusedsafe predicate[2].

```
1  aspect protect_password{
2     advice: in("Login",unbound location uid,unbound string pw)
3        &&on(bound location s)&&target(bound location client)
4        &&continuation(process z){
5        if(element_of(client,{Client1,Client2})&&
6           !beusedsafe(pw,{OUT},{uid},z))
7           terminate;
8        else
9           proceed;
10    }
11 }
```

Listing 6.8: Aspect for Protecting Password

The aspect matches an in action (like the one that receives a login request at Line 5 in Listing 6.4), and checks if the continuation process sends the password only to the user's server (like the action at Line 6) but not to other locations (like the action at Line 8).

The pointcut of this aspect uses the *unbound* modifier for some of its parameters. The unbound modifier means that the variables are not bound to any value before the action is performed. (Note that an in action is used to retrieve a tuple.) When a client performs an in action (Line 5 in Listing 6.4) with a "Login" tag, the pointcut in Listing 6.8 matches it and binds Client1 to both s and client. It also records that variables uid and pw in the aspect are connected to variables userserver and password in the client process. Since the variables in the client do not have values at the beginning of the in action, the variables in the aspect are considered to have potential values that will be stored to those variables in future.

[2]AspectKE don't have similar analysis operators to enforce such kinds of policies.

The body of the advice first checks if the targeted location of in action is one of the clients (at Line 5 in Listing 6.8), and if the password is sent to locations other than the user's server node in the continuation process (at Line 6). Here, the beusedsafe predicate checks, if the continuation process z that uses pw (password) in any out action has uid (userserver) as the destination. If it does not (note the negation operator before the predicate), the aspect terminates the client process. Since Client1 will send the password to Eavesdropper (at Line 8), the aspect will terminate the process at the in action at Line 5.

Note that the predicate can check the condition even though variables pw and uid are not bound when the advice runs. This is because the variables denote the potential values they will be bound to in future. These potential values in the continuation process are collected via interprocedural *data-flow analysis*, a type of static analysis which we explain in details in Chapter 7. Briefly speaking, this analysis can collect data flow information of variables within and among processes. For example, we can detect that userserver, assigned by the in action (at Line 5 in Listing 6.4), will be used not only within the continuation process of the same process (Lines 6, 8, 10 of process clientlogin in Listing 6.4), but also will in the processes spawned by this process, e.g., (at Line 6, 7 and 9 of process clientsendmsg in Listing 6.5).

6.3 Highlight of the Language Features

Although AOP is known as a promising approach to implement separately security concerns (e.g., [WJP02]), we find three problems in the existing approaches when designing and implementing practical programming languages that can enforce the above-mentioned policies in Section 6.2.1.2. Below, we will state these problems and highlight our solutions that have been presented in Section 6.2.3.

6.3.1 Predicting Control- and Data-flows

Many of existing AOP languages including AspectJ cannot apply aspects based on control and data flow from current execution point (or, the *join point*), which are required to implement Security Policies 2 and 3. Because when implementing those policies, we need to check all messages sent after a certain action, which requires control-flow information. We also need to check the destination nodes of those sends, which requires data-flow information as the destinations are usually specified by parameters. In addition, Policy 3 requires data-flow information for passwords to check if the message sends contain passwords.

Most existing AOP languages can only use merely past and current information available at the join point, but not future behavior of a program, in order to trigger execution of aspects. For example, `cflow` [KHH+01], `dflow` [MK03], and `tracematches` [AAC+05] are AOP constructs in AspectJ- like languages that trigger execution of aspects based on calling-context, data-flow, and execution history, respectively, in the *past execution*. Those constructs would be useful to implement some of the security policies like Policy 1, but not so for Policies 2 and 3. A few AOP languages propose mechanisms by which aspects can be triggered by control flow of a program in the future, e.g, `pcflow` [Kic03] and `transcut` [SMH09], however, to use them for enforcing Policies 2 and 3 is difficult, due to their incapability to expose data-flow information in the future.

AspectKE* approach is to perform static control and data-flow analysis of processes to be executed, and provide a set of practical and expressive predicates and functions that extract information on future behavior of a continuation process from the analysis (or the *proceeded* execution, to follow the AOP's terminology).

6.3.2 Ease of Description of Policies

Even though several AOP extensions [AM07, OMB05, CN04, KRH04] offer the means of predicting future behavior, it is not easy to describe security policies in those extended languages because users have to deal with low-level information. For example, Josh [CN04], LogicAJ [KRH04], and SCoPE [AM07] allow users to define a pointcut that uses results of a static program analysis. However, users have to write programs that analyze bytecode or source program, which is not an easy task. Though a library of typical analysis functions might help, currently there are no such libraries available.

The AspectKE* approach provides several high-level predicates and functions that give basic information on a program's future behavior. Users can then easily combine those predicates and functions for implementing security policies, and utilize the analysis results for specifying different security policies.

6.3.3 Combining Static and Dynamic Conditions

In order to implement security policies, we need to check both static and dynamic conditions, which existing approaches cannot support elegantly. For example, consider conformity of the following code fragment with Policy 2 and refer to the steps in Figure 6.1.

1: *receive the server location into* userserver (8)
2: *assign* userserver *to* u
3: *send a message to* u (9)
4: *send a message to* ServerAlice (9')

In order to judge conformity, we need to know, before executing Line 1, the destinations of message sends at Lines 3 and 4 are the same as the value in usersever, which requires both static and dynamic checking. For Line 3, we need to statically analyze the program to determine if u will have the same value as userserver. For Line 4, we need to check that the runtime value of userserver is indeed ServerAlice.

Even in the AOP languages that support static program analyses, programmers have to write static analysis and dynamic condition separately. This will make the aspect definitions redundant, difficult to understand, and hard to maintain.

The AspectKE* approach is to provide a static-dynamic *dual value evaluation mechanism* that can compare both statically and dynamically values available in the join point and parameters to future actions while writing a single comparison expression. When a parameter to a future action is a variable, AspectKE* statically checks data flow from the value at the join point. When the parameter is a constant, it dynamically checks against the runtime value at the join point. With this mechanism, programmers need merely to write one single expression for a comparison, which makes it simpler and easier to maintain security policies. This feature has been explicitly illustrated when enforcing Policy 2 in Section 6.2.3.

6.4 Concluding Remarks

In this chapter we have presented the language design of AspectKE* and illustrated how to enforce security policies, especially the ones based on future behavior to a distributed chat system containing malicious processes. We also highlighted the language features that distinguish AspectKE* from other aspect-oriented programming languages.

In the next chapter we will explain how runtime system is developed to support the language design. We will focus on discussing how static analysis is performed and integrated into aspects efficiently.

Chapter 7

AspectKE*: Implementation

We implemented a prototype compiler and runtime system for AspectKE*, which are publicly available[1]. The compiler is written in 1618 lines of code on top of the ANTLR and StringTemplate frameworks. The runtime system is a Java package consisting of an analyzer and an interpreter. It is built on top of the Klava package [BDP02], with 6506 lines of Java code.

In this chapter, Section 7.1 overviews our implementation, and highlights the mechanism that efficiently evaluates program analysis functions and predicates, and the dual value evaluation mechanism that combines evaluation of static and runtime information. Section 7.2 presents the AspectKlava runtime system, which can be used to program an aspect-oriented tuple space system in Java. AspectKlava also serves as the runtime system of AspectKE*. Section 7.3 presents the interprocedural data-flow analysis on Java bytcode of AspectKlava processes, and how it provides support for runtime evaluation of program analysis predicates and functions.

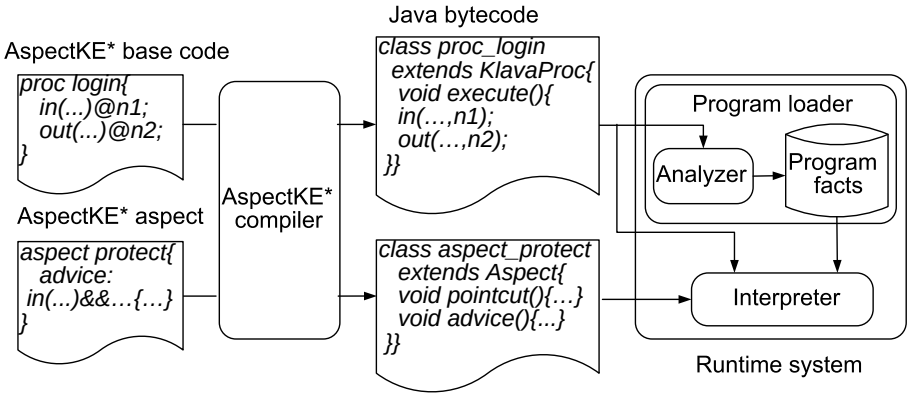

Figure 7.1: Overview of the Implementation

7.1 Overview of the System

Figure 7.1 shows an overview of our implementation. The compiler generates a Java class for each node and process defined in the given base code. At this stage, aspects are translated into Java classes independently from the base code. The weaving [MKD03,PGA02] process is carried out by the interpreter at runtime so that new aspects can be add to a running system without restarting.

The analyzer extracts, from a set of Java bytecode, program facts via a context-insensitive interprocedural data-flow analysis on Java bytecode that is implemented on top of ASM [BLC02]. The interpreter uses the program facts for evaluating program analysis predicates and functions. Sections 7.1.1 and 7.1.2 explain the mechanism at conceptual level, while Section 7.2 and 7.3 elaborate with more technical materials for a deep understanding the mechanism.

The architecture fits well the execution model of Klava which supports code mobility. Creation of a process at a remote node, e.g., eval(process p)@node, in Klava, sends only a Java class file that implements the process p to node. As the source code of the mobile process is not runtime available to a remote node, we let the analyzer perform program analysis on the Java bytecode format of a process instead. In the above example, the bytecode instructions of process p will be analyzed, before its execution, when it arrives at node .

[1]http://www.graco.c.u-tokyo.ac.jp/ppp/projects/aspectklava.en

7.1.1 Efficient Evaluation of Program Analysis Predicates and Functions

In principle, evaluation of program analysis functions and predicates would take place at runtime as they appear inside an advice body. A naive implementation that performs runtime program analysis (as the AspectKE execution model) is, however, not practical due to unacceptable runtime overhead.

Our implementation avoids the runtime overhead by separating execution of the program analysis from evaluation of program analysis predicates and functions. When the runtime system loads the definition of a process, it analyzes the definition and extracts *program facts* for each (shadow [MKD03] of) action in the process. Later on, the advice body uses the program facts for evaluating program analysis predicates and functions.

Note that our approach analyzes each process definition only once no matter how many aspects are applied to (any) actions in the process, and no matter how many program analysis predicates and functions are used and evaluated. In this way we minimize the overhead of the expensive program analysis.

A program fact is not a cached result of a program analysis predicate or function, but is more primitive information about the program. Each program fact (of an action in a process) consists of the following three data:

- the *actions* (e.g., `out`) that will be executed by the remaining process,
- the *destination locations* specified as the destinations of actions in the remaining process, and
- predicted data-flow information called *pdflow* that represents, for each parameter of an action, where it will be used in actions that use the same parameter in the remaining process.

7.1.2 Dual Value Evaluation Mechanism

Our language supports static and dynamic conditions in one expression by binding both static and runtime information to each variable in a pointcut. Additionally, a runtime value can be compared with values obtained by static analysis. Here, we illustrate AspectKE*'s underlying dual value evaluation mechanism.

A predicted data-flow (*pdflow*) is represented by a set of *useID*s, which identifies the positions of the parameters in a process. In other words, each variable used in an action is represented by a *useID*, and is associated with a set of *useID*s that

predicts its data flow in the future. *Destination locations* are also represented by *useID*s because they are part of the future data flow.

A *useID* is a pair of integers $\langle actionIx, paramIx \rangle$, where *actionIx* identifies an action in a process, and *paramIx* indicates the parameter position of a variable in the action.

The *useID*s are also used to check equality between a value captured from a join point u and a value obtained from program analysis functions x (e.g., <x==uid> at Line 6 in Listing 6.7) where x represents a variable in a future action. In this case, an equality expression $<u == x>$ judges if, when the process resumes, the variables represented by u and x will potentially have the same value.

In the following paragraphs, we explain how the condition at Line 6 in Listing 6.7 is evaluated with respect to process clientsendmsg in Listing 6.5 (except for the last eval action) by using the *useID*s.

Labeling action parameters at compile-time. The compiler labels each parameter of the action in a process with either a unique *useID* or a constant when translating the AspectKE* source code to Java bytecode. The labeled actions are shown below.

```
1 | in("Msg"^"Msg", userserver^⟨1,2⟩, friendserver^⟨1,3⟩, text^⟨1,4⟩)
2 |                                                  @self^⟨1,0⟩;
3 | out("Msg"^"Msg", friendserver^⟨2,2⟩, text^⟨2,3⟩, self^⟨2,4⟩)
4 |                                              @userserver^⟨2,0⟩;
5 | out(userserver^⟨3,1⟩, friendserver^⟨3,2⟩, text^⟨3,3⟩)
6 |                                  @Eavesdropper^Eavesdropper;
```

If a constant is given directly as a parameter, it is labeled with the constant itself. Otherwise, a parameter is labeled with a *useID* that is unique within the process.

Extracting the program facts at load-time. When a node loads a process at runtime, the analyzer extracts the program facts for each action in the process and those processes under the control flow of this process.

The flow information *pdflow* for the userserver at in action, namely $pdflow_{\mathsf{in}}$ contains $\{\langle 1,2\rangle, \langle 2,0\rangle, \langle 3,1\rangle\}$ because userserver is used as the destination of the first out action and the first parameter of the second out action. *pdflow*s for other parameters and those in the two out actions are created similarly.

The destination locations are computed with the help of *pdflow*. The analyzer first collects the set of *useID*s and constants used as the destinations of actions,

and then replaces each *useID* in the set with the first *useID* in the *pdflow* that contains it. For example, the destination location for the in action, namely $ploc_{\mathsf{in}}$, becomes {(OUT,⟨1, 2⟩), (OUT,Eavesdropper)}. This is because the *useID* for the first out action's destination is ⟨2, 0⟩, which belongs to $pdflow_{\mathsf{in}}$ whose first element is ⟨1, 2⟩.

Runtime pointcut matching and equality evaluation. When a node executes the in action at Line 6 in Listing 6.5, the pointcut in aspect protect_message in Listing 6.7 matches, and the condition forall(x,targeted({OUT},z)) <x==uid> is checked. Here, uid binds two things: one is a value of either ServerAlice or ServerBob, and the other is the *useID* of the second parameter of this in join point action, namely $useID_{\mathsf{uid.in}}$, i.e., ⟨1, 2⟩. z binds the continuation process which yields, for targeted({OUT},z), {⟨1, 2⟩,Eavesdropper} by simply referencing $ploc_{\mathsf{in}}$.

The interpreter checks for each element x in {⟨1, 2⟩,Eavesdropper} whether x is equal to uid, by comparing the uid's value (ServerAlice or ServerBob) and *useID* ($useID_{\mathsf{uid.in}}$=⟨1, 2⟩). When x is ⟨1, 2⟩, the equality holds because $useID_{\mathsf{uid.in}}$ is used. When x is Eavesdropper, the equality fails when it is compared against a runtime value.

7.2 AspectKlava Runtime System

In this section, we present the runtime system of AspectKE*: the AspectKlava Java package. AspectKlava is built on top of the Klava package [BDP02], a Java implementation of the KLAIM process calculus [DFP98]. Klava offers intuitive Java interfaces, classes and methods to program the KLAIM programming model, which supports all core programming concepts from KLAIM, such as nodes, processes and actions. AspectKlava focuses on extending the existing Klava packages to support the additional aspect-oriented programming paradigm. In particular, it provides a sub-package that implements an analyzer which performs static analysis on the bytecode format of AspectKlava processes. This sub-package is built on top of the ASM [BLC02] analysis framework. ASM is a light-weight all-purpose Java bytecode manipulation and analysis framework.

The AspectKlava runtime system requires the following Java packages: AspectKlava, Klava, IMC [BDNF+04] [2] and ASM. From AspectKE* source code, the AspectKE* compiler first generates a Java source program that relies on

[2] IMC is a Java framework for implementing distributed applications possibly with code mobility, the latest Klava is built on top of it.

these packages, which then can automatically be compiled into executable Java bytecode.

Below, we introduce the design of the AspectKlava package for user to better understand the key techniques used for implementing the runtime system for AspectKE*, to better understand the program generated by AspectKE* compiler, to directly program the AspectKE programming model in Java, or extend the package with other advanced functionality. Some of the classes introduced will be analyzed in the following Section 7.3, where we exhibit how static analysis is developed and integrated in AspectKlava.

7.2.1 Tuples and Tuple Spaces

AspectKlava directly utilizes the implementation of tuple and tuple space in Klava. Here we briefly introduce its basic usage.

In an action, the operated tuple is formed by fields that are either an actual field (e.g. constant value) or a formal field (e.g. non-initialized variable). Note that two additional types of fields can be used in aspects, which will be explained shortly.

The class Tuple includes methods for tuple-relevant operations, such as creation of tuples, tuple matching etc. To create a Tuple, an easy way is to create an empty tuple and then append different fields in sequence.

Actions can use pattern-matching to select tuples from a tuple space: Two tuples match each other if they have the same number of fields and each corresponding field matches: formal fields match any actual field of the same type, and two actual fields match if they have the same actual value.

The interface TupleItem in Klava package is used to define concrete types of tuple fields.

```
public interface TupleItem extends java.io. Serializable {
    public boolean isFormal (); //whether this item is formal
    public void setValue(Object o); //to set the value of this item
    public boolean equals(Object o); //whether they are equal
    public Object duplicate (); //duplicate an item
}
```

The package Klava provides several wrapper classes for standard data types such as KString (string type), LogicalLocality (location type) and KlavaProcessVar (process type) that implement this interface.

Example 7.1 Listing 7.1 illustrates the basic usage of tuple.

```
1   KString f1 = new KString("Location");        // actual declaration
2   LogicalLocality f2 = new LogicalLocality(); // formal declaration
3   Tuple t1 = new Tuple();
4   t1.add(f1);
5   t1.add(f2);
6
7   Tuple t2 = new Tuple();
8   t2.add(new KString("Location"));
9   t2.add(new LogicalLocality("Alice"));
10
11  t2.match(t1);
12  System.out.println("s now is: "+f2);
```

Listing 7.1: Tuple and Pattern-Matching (in AspectKlava)

At Line 11, variable f2 (declared at Line 2) will be bound with Alice through pattern-matching. □

Tuple spaces store tuples. Interface TupleSpace defines basic method for tuple operations such as out, in action etc. Class TupleSpaceVector is provided by Klava as the implementation of TupleSpace, which is also internally used in the AspectKlava.

7.2.2 Localities

AspectKlava uses similar infrastructure to Klava to handle localities which refer to nodes of a net. There are two kinds of localities:

- physical localities (class PhysicalLocality) identify nodes in a real network (e.g. ip address and port number).
- logical localities are symbolic names for nodes (aliases for network resources). Class LogicalLocality is used in the main program, while ALogicalLocality is used in the aspect program.

Each node registers to a name server a unique logical locality for processes running on those nodes. The name server automatically translates the logical localities into physical localities when needed. For example, when a tuple is outputted to a node that is identified by its logical locality, this tuple can always be delivered to the actual physical locality of the destination node.

7.2.3 Nodes

In AspectKlava, each node extends class AKlavaNode and defines its own tuple space and processes. AKlavaNode itself extends class KlavaNode from the Klava package, and provides only one constructor, to force a user to register with its own symbolic name (i.e., logical locality) to a name server.

```
public AKlavaNode(PhysicalLocality server, LogicalLocality logicalLocality )
                                                    throws Exception
```

When creating a node, a pool of aspects (Vector<Aspect>) are initialized and bound to the node. The aspects will be furthered attached to the node and monitor processes executing on it. The process could be either locally instantiated by this node or remotely instantiated by other nodes.

The actions in the AspectKE programming model are normally executed by processes (which shall be monitored by aspects). We provide several methods in nodes which are not controlled by aspects, denoted with symbol "a" in front of the actions.

```
public void aout(Tuple tuple, LogicalLocality destination ) throws KlavaException;
public void ain(Tuple tuple, LogicalLocality destination ) throws KlavaException;
public void aread(Tuple tuple, LogicalLocality destination ) throws KlavaException;
public void aeval(AKlavaProcess aklavaProcess, LogicalLocality destination )
                                                    throws KlavaException;
public void anewloc( LogicalLocality freshlocality ) throws KlavaException;
```

Methods aout, ain, aread take parameters tuple and destination, which are useful for transmitting tuples among different nodes; method aeval is useful for executing a process AklavaProcess on a local or remote site. Method anewloc takes a parameter freshlocality which is bound with the actual logical locality of a newly created node.

These methods are intended to be used for privileged users to perform special tasks, while their localized version such as:

```
public void aout(Tuple tuple) throws KlavaException;
public void aeval(AKlavaProcess aklavaProcess) throws KlavaException;
```

are used for initializing tuple space and processes when creating a node. Method aeval initiates a process (subclass of AklavaProcess) to itself, while method aout can be used to initialize tuples to its own tuple space.

As already mentioned earlier, a special node for name server has to be set up to do the name resolution job in AspectKlava net. We use LogicalNet(a subclass of KlavaNode, provided by Klava package) to create this name server node. All

other nodes (subclass of AKlavaNode) can use the constructor shown above to automatically register to this server when instantiating the node.

Example 7.2 This example will illustrate how to construct a chat application net presented in Section 6.2 (Listing 7.2), and will show functional components of the net: node definitions of Node_ServerAlice (Listing 7.3) and Node_Client1 (Listing 7.4).

```
1   PhysicalLocality server = new PhysicalLocality("tcp-127.0.0.1:9999");
2   /* Initialize name server node */
3   new LogicalNet(server);
4
5   LogicalLocality loc;
6
7   loc = new LogicalLocality("ServerAlice");
8   /* Initialize node ServerAlice, connect it to name server node */
9   new Node_ServerAlice(server, loc);
10
11  loc = new LogicalLocality("Client1");
12  /* Initialize node Client1, connect it to name server node */
13  new Node_Client1(server, loc);
```

Listing 7.2: Part of the Net for Chat Application (in AspectKlava)

At Listing 7.2 Line 1, following the syntax of IMC session identifiers [BDNF+04], server is declared and defined as a physical locality: tcp-127.0.0.1:9999, which indicates tcp is the protocol used for the server, 127.0.0.1 is its ip address, 9999 is the port number the server listens to. In AspectKlava, we always specify physical localities with format: tcp-<ip>:<port>. Line 3 creates the name server node for our chat application, while Lines 9 and 13 create two function nodes for the application.

Note that Listing 7.2 shows a program that instantiates the chat application only at one computer, which is a special case as different nodes can start the application at different computers.

```
1  public class Node_ServerAlice extends AKlavaNode {
2      public Node_ServerAlice(PhysicalLocality server, LogicalLocality logicalLocality )
3              throws Exception {
4          super(server, logicalLocality );
5          startDB();
6          startProc ();
7      }
8
9      public void startDB() throws KlavaException{
10         Tuple t1 = new Tuple();
11         t1.add(new KString("Password"));
12         t1.add(new KString("abc123"));
```

```
13          aout(t1);
14      }
15
16      public void startProc() throws KlavaException{
17          AKlavaProcess p_userlogin = new Process_userlogin(new LogicalLocality("ServerAlice"));
18          aeval(p_userlogin);
19      }
20  }
```

Listing 7.3: Node ServerAlice (in AspectKlava)

```
1   public class Node_Client1 extends AKlavaNode {
2       public Node_Client1(PhysicalLocality server, LogicalLocality logicalLocality)
3               throws Exception {
4           super(server, logicalLocality);
5           startDB();
6           startProc();
7       }
8
9       public void startDB() throws KlavaException{
10      }
11
12      public void startProc() throws KlavaException{
13          AKlavaProcess p_clientlogin = new Process_clientlogin(new LogicalLocality("Client1"),
14              new LogicalLocality("Console1"));
15          aeval(p_clientlogin);
16      }
17  }
```

Listing 7.4: Node Client1 (in AspectKlava)

In Listing 7.3 and 7.4, both classes Node_ServerAlice and Node_Client1 extend AKlavaNode, which can be directly generated from AspectKE* node definitions. For example, Node_Client1 can be obtained from Listing 6.3 by using the AspectKE* compiler. In these nodes, the startDB method initializes the tuple space of a node. For example, Lines 10-13 in Listing 7.3 output a tuple regarding the password to Node_ServerAlice; while Lines 9-10 in Listing 7.4 declare an empty tuple space for Node_Client1; the startProc method is used to instantiate a process on a node. For example, Lines 12-16 in Listing 7.4 start a Process_clientlogin at Node_Client1. □

7.2.4 Processes

Processes consist of actions that operates on tuples. Each process extends the abstract class AKlavaProcess, which has to define the following abstract method:

```
public abstract void AexecuteProcess() throws KlavaException;
```

In order to define actions for a process, the following primitive methods from class AKlavaProcess can be invoked and used in this abstract method. These methods (actions) will be monitored by aspects assigned to the node where the process is currently running:

```
public void out(Tuple tuple, LogicalLocality destination) throws KlavaException;
public void in(Tuple tuple, LogicalLocality destination) throws KlavaException;
public void read(Tuple tuple, LogicalLocality destination) throws KlavaException;
public void eval(AKlavaProcess aklavaProcess, LogicalLocality destination)
                    throws KlavaException;
public void newloc(LogicalLocality freshlocality) throws KlavaException;
```

Note that Klava has two types of processes: ordinary processes (subclass of KlavaProcess) and privileged processes. An ordinary process has mobile capability (to be executed at remote nodes) and could perform all types of actions except for newloc (newloc can only be performed by privileged processes). Privileged processes, on the other hand, don't have mobile capability – this is Klava design decision. In AspectKlava, we follow the AspectKE programming model and do not differentiate privileged processes from ordinary processes. AKlavaProcess extends KlavaProcess and equips additional newloc functionality. Therefore subclasses of AKlavaProcess can perform all types of actions and also preserve mobile capability.

We have briefly mentioned in Section 7.1.2 that our dual value evaluation mechanism requires to label parameters of actions at compile-time, and assigns each parameter in a process with a unique *useID* (a pair of integers $\langle actionIx, paramIx \rangle$). *paramIx* can be internally calculated according to the relative position of the parameter in an action. Here we have to explicitly assign *actionIx* to each action in a process when compiling from AspectKE* source code. Therefore we provide another version of the five action methods where an additional parameter is added. For example,

```
public void out(Tuple tuple, LogicalLocality destination, int internalnum)
                                                    throws KlavaException;
```

this method will in the end invoke the first version of out method which does not contain the third parameter.

Example 7.3 Listing 7.5 defines process Process_clientsendmsg (compiled from Listing 6.5). Lines 22, 29, 34 and 37 declare four actions with assigned *actionIx*. We will describe how to analyze the bytecode instructions of this process in Section 7.3.

```
1  public class Process_clientsendmsg extends AKlavaProcess {
```

```
2       LogicalLocality  self ;
3       LogicalLocality  userserver ;
4       LogicalLocality  console;
5
6       public Process_clientsendmsg( LogicalLocality  self , LogicalLocality  userserver ,
7               LogicalLocality  console ) throws KlavaException {
8           this . self =self;
9           this . userserver =userserver;
10          this . console=console;
11      }
12
13      public void AexecuteProcess() throws KlavaException {
14          LogicalLocality  friendserver  = new LogicalLocality ();
15          KString text  = new KString();
16
17          Tuple t1  = new Tuple();
18          t1.add(new KString("Msg"));
19          t1.add( userserver );
20          t1.add( friendserver );
21          t1.add(text);
22          in(t1, self ,1);
23
24          Tuple t2  = new Tuple();
25          t2.add(new KString("Msg"));
26          t2.add( friendserver );
27          t2.add(text);
28          t2.add( self );
29          out(t2, userserver ,2);
30
31          Tuple t3  = new Tuple();
32          t3.add( userserver );
33          t3.add( friendserver );
34          out(t3,new  LogicalLocality ("Eavesdropper"),3);
35
36          AKlavaProcess p_clientsendmsg = new Process_clientsendmsg(self,userserver, console );
37          eval(p_clientsendmsg, self ,4);
38      }
39  }
```

Listing 7.5: Process clientsendmsg (in AspectKlava)

□

7.2.5 Aspects

Data types in Aspects Here we introduce data types used in aspects, which implement the interface ATupleItem and extend the main program's corresponding data type. For example, KString is used in the main program, which implements TupleItem and can represent two fields status: actual and formal fields. AKString is used in aspects, which extends class KString and implements interface ATupleItem. Using the interface ATupleItem, AKString can represent four different fields status: actual (constant), formal (bound variable), unbound variable and don't-care (Please refer for the exact meaning of these status to our development of AspectKE or AspectKE*).

The interface ATupleItem is declared as follows:

```
public interface ATupleItem extends TupleItem {
    public boolean isVariable (); //whether the item is unbound variable field
    public boolean isNotcare (); //whether the item is don't−care field
    public boolean isConstant (); //whether this item is a constant field
    public boolean compareConstant(Object o); //compare the value of this item
                                              with Object o
    public Object convert (); //convert itself to its counterpart in TupleItem
                              when possible
    public void setPosition (int pos); // set parameter position (paramIx)
    public int getPosition (); // get parameter position (paramIx)
}
```

In ATupleItem, the setPosition and getPosition methods handles *paramIx* (part of the *useID*, introduced in Section 7.1), which is used for developing static analyses and integrating the analysis results into aspects.

AspectKlava has implemented data types AKString, ALogicalLocality, AKlavaProcessVar. Note that when defining a new aspect data type, all methods declared in ATupleItem have to be implemented, and methods defined in TupleItem have to be overridden even though the methods can be inherited from a superclass.

Example 7.4 Listing 7.6 illustrates how to declare data types used in aspects by using different class constructors.

```
1  AKString k = new AKString("foo");     // constant declaration
2  AKString s = new AKString();          // formal (bound variable) declaration
3  AKString t = new AKString(true);      // unbound variable declaration
4  AKString p = new AKString(false);     // don't−care declaration
```

Listing 7.6: Data Type Declaration in Aspect (in AspectKlava)

Lines 1-4 illustrate how to declare the four status of the data type AKString. □

Aspects In AspectKlava, each node maintains its own copy of aspects (to be called aspectpool), which can be obtained from class AspectMonitor that handles all aspect-related issues. To enforce aspects to a node, we have to explicitly invoke the AspectMonitor.loadAspects() method at the start of main program.

An aspect consists of a pointcut and an advice. Each aspect in AspectKlava has to implement the abstract class Aspect. There are four attributes defined in this abstract class:

```
public ActionType action;
public ALogicalLocality source;
public ALogicalLocality destination ;
public Vector<ATupleItem> parameters;
```

These attributes should be initialized and defined in the abstract method pointcut, and they can be used within a concrete advice by implementing the abstract method advice.

```
public abstract void pointcut();
public abstract void advice() throws AspectKlavaException;
```

There are several methods (primitives) which can be used inside method advice:

```
public boolean test(Tuple tuple, ALogicalLocality destination) throws AspectKlavaException;
public boolean equal(ATupleItem o1, ATupleItem o2);
public HashSet<ActionType> getActionSet_X() throws AspectKlavaException;
public HashSet<ActionType> getActionSet_Y() throws AspectKlavaException;
public HashSet<ALogicalLocality> getLocSet_X(HashSet<ActionType> actiontypes)
    throws AspectKlavaException;
public HashSet<ALogicalLocality> getLocSet_Y(HashSet<ActionType> actiontypes)
        throws AspectKlavaException;
public boolean isFV_X(ATupleItem u, HashSet<ActionType> actiontypes)
                throws AspectKlavaException;
public boolean isFV_Y(ATupleItem u, HashSet<ActionType> actiontypes)
                                throws AspectKlavaException;
public boolean isFVSafe_X(ATupleItem u, HashSet<ATupleItem> locs,
                HashSet<ActionType> actiontypes) throws AspectKlavaException;
public boolean isFVSafe_Y(ATupleItem u, HashSet<ATupleItem> locs,
                HashSet<ActionType> actiontypes) throws AspectKlavaException;
```

Method test checks whether a tuple is stored in the tuple space of a certain destination. Method equal tests whether two data are equal or not. Two data are equal if they have the same constant values (these constant values can be obtained at runtime from the join point, or at loadtime from the program facts), or if they are both variable and have data flow between them. The other methods are program analysis predicates and functions, whose meaning has been specified in Table 6.3. Although the names are slightly different. Methods ending with

_X return analysis results of the continuation process of the current action (e.g., out(...)@X); while methods ending with _Y return analysis results of the remote evaluation process (e.g., eval(Y)@X).

In addition, class Aspect refers to the node where this aspect is currently located. The reference is important because method test relies on the network capability of nodes.

Example 7.5 Here we show the aspect for protecting message, which is compiled from Listing 6.8.

```
1  public class Aspect_protect_message extends Aspect {
2      public Aspect_protect_message(String aspectname){
3          super(aspectname);
4      }
5      ALogicalLocality uid;
6      ALogicalLocality s;
7      ALogicalLocality client;
8
9      public void pointcut() {
10         this.setType(ActionType.In);
11         s = new ALogicalLocality(true);
12         this.setSource(s);
13         client = new ALogicalLocality(true);
14         this.setDestination(client);
15         this.appendParameter(new AKString("Msg"));
16         uid = new ALogicalLocality(true);
17         this.appendParameter(uid);
18         this.appendParameter(new ALogicalLocality(false));
19         this.appendParameter(new AKString(false));
20     }
21
22     public boolean advice() throws AspectKlavaException {
23       HashSet temp_s_1 = new HashSet();
24       temp_s_1.add(new ALogicalLocality("Client1"));
25       temp_s_1.add(new ALogicalLocality("Client2"));
26
27       HashSet temp_s_2 = new HashSet();
28       temp_s_2.add(ActionType.Out);
29
30       boolean b_1_2 = temp_s_1.contains(client);
31       boolean b_1_4 = true;
32       for(Object x: getLocSet_X(temp_s_2)){
33           boolean b_2_1 = equal(x,uid);
34           b_1_4 = b_2_1;
35           if(b_1_4==false) break;
36       }
37       boolean b_1_3 = !b_1_4;
```

```
38          boolean b_1_1 = (b_1_2 && b_1_3);
39          if(b_1_1){
40          return  false;
41          }
42          return  true;
43        }
44  }
```

Listing 7.7: Aspect for Protecting Chat Information (in AspectKlava)

In method pointcut, we initialize the essential elements (action, source, destination, parameters) in the pointcut, which are declared in superclass Aspect.

When executing method advice, all attributes defined in the pointcut are updated with the actual value from the join point action. Line 32 uses program analysis function getLocSet_X. □

7.3 Static Analysis of Process in AspectKlava

In last section, we reviewed the design and usage of core programming concepts in AspectKlava. In this section, we will focus on its static analysis module and present how static analysis has been developed and used in AspectKlava.

7.3.1 The Bytecode Instructions of a Process

As shown in Figure 7.1, our analyzer performs directly static analysis on *Java bytecode*, i.e., the form of instructions that Java virtual machine executes [LY99]. This provides the possibility to enforce security policies to processes where the source code (AspectKE*/Java) is difficult to obtain – a common scenario in a distributed, mobile system.

First let us take a look at the disassembled bytecode of a Java class file. Listing 7.8 shows the bytecode representation of the cliendsendmsg process, whose Java source code is shown in Listing 7.5 (compiled from Listing 6.5). Note that for better readability, the bytecode program listed is slightly annotated compared with a standard Java bytecode program [LY99].

Lines 5-9, Lines 13-32 and Lines 36-157 in Listing 7.8 show attribute declarations, process constructor and method AexecuteProcess, which corresponds with Lines 2-4, Lines 6-11 and Lines 13-38 in Listing 7.5, respectively. Each Java statement in Listing 7.5 maps a sequence of instructions between two labels in

```
1   public class Process_clientsendmsg extends AKlavaProcess {
2
3     // compiled from: Process_clientsendmsg.java
4
5     LogicalLocality self
6
7     LogicalLocality userserver
8
9     LogicalLocality console
10
11    // construction method of Process_clientsendmsg
12
13    public <init>(LogicalLocality, LogicalLocality, LogicalLocality) :
14                                  void throws KlavaException
15      L0
16        ALOAD 0: this
17        INVOKESPECIAL AKlavaProcess.<init>() : void
18      L1
19        ALOAD 0: this
20        ALOAD 1: self
21        PUTFIELD Process_clientsendmsg.self : LogicalLocality
22      L2
23        ALOAD 0: this
24        ALOAD 2: userserver
25        PUTFIELD Process_clientsendmsg.userserver : LogicalLocality
26      L3
27        ALOAD 0: this
28        ALOAD 3: console
29        PUTFIELD Process_clientsendmsg.console : LogicalLocality
30      L4
31        RETURN
32      L5
33
34    // AexecuteProcess method of Process_clientsendmsg
35
36    public AexecuteProcess() : void throws KlavaException
37      L0
38        NEW LogicalLocality
39        DUP
40        INVOKESPECIAL LogicalLocality.<init>() : void
41        ASTORE 1
42      L1
43        NEW KString
44        DUP
45        INVOKESPECIAL KString.<init>() : void
46        ASTORE 2
47      L2
48        NEW Tuple
49        DUP
50        INVOKESPECIAL Tuple.<init>() : void
51        ASTORE 3
52      L3
53        ALOAD 3: t1
54        NEW KString
55        DUP
56        LDC "Msg"
57        INVOKESPECIAL KString.<init>(String) : void
58        INVOKEVIRTUAL Tuple.add(Object) : void
59      L4
60        ALOAD 3: t1
61        ALOAD 0: this
62        GETFIELD Process_clientsendmsg.userserver : LogicalLocality
63        INVOKEVIRTUAL Tuple.add(Object) : void
64      L5
65        ALOAD 3: t1
66        ALOAD 1: friendserver
67        INVOKEVIRTUAL Tuple.add(Object) : void
68      L6
69        ALOAD 3: t1
70        ALOAD 2: text
71        INVOKEVIRTUAL Tuple.add(Object) : void
72      L7
73        ALOAD 0: this
74        ALOAD 3: t1
75        ALOAD 0: this
76        GETFIELD Process_clientsendmsg.self : LogicalLocality
77        ICONST_1
78        INVOKEVIRTUAL Process_clientsendmsg.in
79                (Tuple, LogicalLocality, int) : void
80      L8
81        NEW Tuple
82        DUP
83        INVOKESPECIAL Tuple.<init>() : void
84        ASTORE 4
85      L9
86        ALOAD 4: t2
87        NEW KString
88        DUP
89        LDC "Msg"
90        INVOKESPECIAL KString.<init>(String) : void
91        INVOKEVIRTUAL Tuple.add(Object) : void
92      L10
93        ALOAD 4: t2
94        ALOAD 1: friendserver
95        INVOKEVIRTUAL Tuple.add(Object) : void
96      L11
97        ALOAD 4: t2
98        ALOAD 2: text
99        INVOKEVIRTUAL Tuple.add(Object) : void
100     L12
101       ALOAD 4: t2
102       ALOAD 0: this
103       GETFIELD Process_clientsendmsg.self : LogicalLocality
104       INVOKEVIRTUAL Tuple.add(Object) : void
105     L13
106       ALOAD 0: this
107       ALOAD 4: t2
108       ALOAD 0: this
109       GETFIELD Process_clientsendmsg.userserver : LogicalLocality
110       ICONST_2
111       INVOKEVIRTUAL Process_clientsendmsg.out(Tuple,LogicalLocality,int) : void
112     L14
113       NEW Tuple
114       DUP
115       INVOKESPECIAL Tuple.<init>() : void
116       ASTORE 5
117     L15
118       ALOAD 5: t3
119       ALOAD 0: this
120       GETFIELD Process_clientsendmsg.userserver : LogicalLocality
121       INVOKEVIRTUAL Tuple.add(Object) : void
122     L16
123       ALOAD 5: t3
124       ALOAD 1: friendserver
125       INVOKEVIRTUAL Tuple.add(Object) : void
126     L17
127       ALOAD 0: this
128       ALOAD 5: t3
129       NEW LogicalLocality
130       DUP
131       LDC "Eavesdropper"
132       INVOKESPECIAL LogicalLocality.<init>(String) : void
133       ICONST_3
134       INVOKEVIRTUAL Process_clientsendmsg.out(Tuple,LogicalLocality,int) : void
135     L18
136       NEW Process_clientsendmsg
137       DUP
138       ALOAD 0: this
139       GETFIELD Process_clientsendmsg.self : LogicalLocality
140       ALOAD 0: this
141       GETFIELD Process_clientsendmsg.userserver : LogicalLocality
142       ALOAD 0: this
143       GETFIELD Process_clientsendmsg.console : LogicalLocality
144       INVOKESPECIAL Process_clientsendmsg.<init>(LogicalLocality,LogicalLocality,
145                                                  LogicalLocality) : void
146       ASTORE 6
147     L19
148       ALOAD 0: this
149       ALOAD 6: p_clientsendmsg
150       ALOAD 0: this
151       GETFIELD Process_clientsendmsg.self : LogicalLocality
152       ICONST_4
153       INVOKEVIRTUAL Process_clientsendmsg.eval(AKlavaProcess,LogicalLocality,
154                                                int) : void
155     L20
156       RETURN
157     L21
158   }
```

Listing 7.8: Bytecode Instructions of Process clientsendmsg

Instr ::= ICONST_i | BIPUSH | SIPUSH | LDC | NEW | CHECKCAST | PUTFIELD | GETFIELD | INVOKESPECIAL | INVOKEVIRTUAL | DUP | ASTORE | ALOAD | RETURN

Table 7.1: Instructions Used in Process

Listing 7.8. For example, Line 14 of Listing 7.5 instantiates a new logical locality, whose corresponding (four) bytecode instructions are listed between Label L0 and Label L1 (Lines 38-41).

The meaning of each instruction is defined in [LY99]. The instructions used to form a AspectKlava Process is summarized in Table 7.1 (The parameters of these instructions are not shown). Our analyzer understands and simulates the execution of these instructions, and collects program facts (static information) of a process from its bytecode instructions.

7.3.2 The Data-Flow Analysis of Bytecode Instructions

Data-flow analysis [NNH05] is one of the main approaches for program analysis. It is used to gather information about the possible set of values calculated at various points in a computer program. Different data-flow analyses can be developed for different analysis problems. The aim is to collect appropriate and sufficient information from a program to answer the analysis problems. Classical data-flow analyses include available expressions analysis, reaching definitions analysis, very busy expressions analysis and live variables analysis [NNH05].

In AspectKlava, we develop our own data-flow analysis on bytecode instructions of processes, and the analysis will collect program facts for supporting the evaluation of program analysis predicates and functions.

7.3.2.1 Control Flow Graph of the Analysis

In data-flow analysis, a program can be considered as a graph, i.e, the program's control flow graph (CFG): the nodes are the elementary blocks and the edges describe how control might pass from one elementary block to another.

For example, Figure 7.2 shows the control flow graph of instructions (between Line 106 till Line 111 in Listing 7.8, which is equivalent to Line 29 in Listing 7.5) for an out action. In the figure, each node (basic block) contains one instruction that has unique predecessor and successor nodes. The data-flow analysis can be performed by following the sequential path and collect program facts (analysis results) of the process from these instructions.

For a more complex case, Figure 7.3 shows the control flow graph of instructions (from Line 148 till 154 in Listing 7.8, which is equivalent to Line 37 in Listing 7.5) for an eval action. When analyzing the instruction in block labeled with 153-154, to compute program facts based on this instruction, we have to obtain and integrate the program facts from its parameter process (of the eval action). Additional edges (flows to the start, and from the end of class Process_clientsendmsg) are drawn to indicate these extra control flow. The data-flow analysis is performed by following the paths (not only sequential) that covers all extra edges.

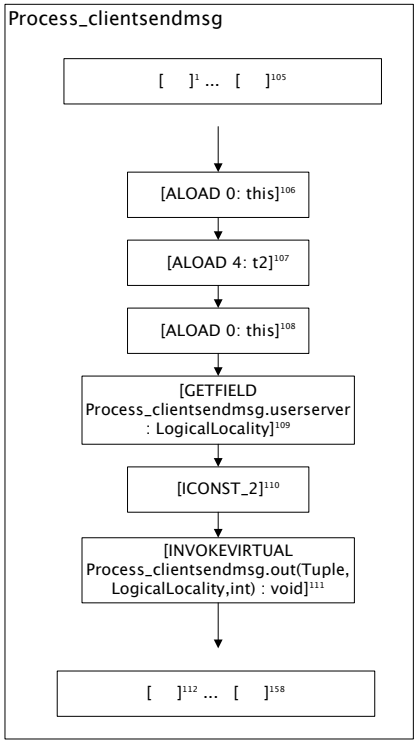

Figure 7.2: Instructions of **out** action (in clientsendmsg)

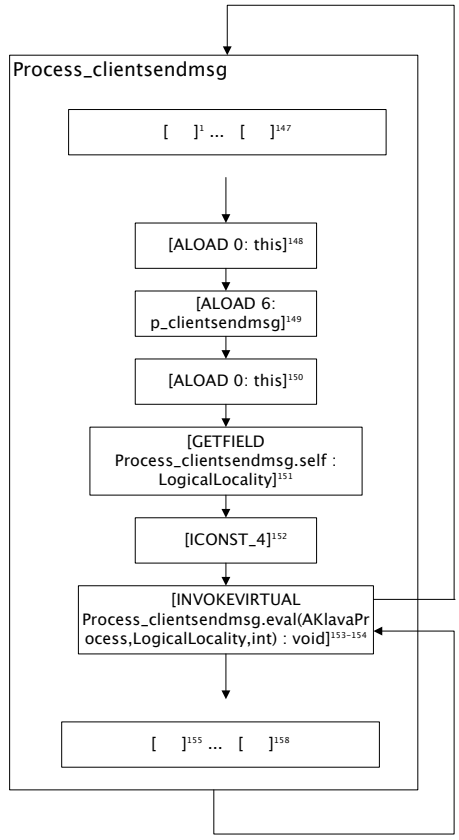

Figure 7.3: Instructions of **eval** action (in clientsendmsg)

7.3.2.2 Forward Flow-Sensitive Analysis

As data-flow analysis aims at obtaining particular information at each point in a program, it is normally to obtain such information at the boundaries of basic blocks. From that it would be easier to compute the information we are interested in.

Type of Analysis There are basically two types of analysis: forward analysis and backward analysis. In a *forward analysis*, the analysis is performed by following the path as a real program normally does, and the exit state of a block is a function of the block's entry state. In a *backward analysis*, the analysis is performed backwards from the end of a program to the beginning of a program, and the entry state of a block is a function of the block's exit state. These functions are called *transfer functions*, as they change the state of variables in a block.

Another view on this is that a program is a *transition system*. Nodes represent blocks and each block has a transfer function associated with it and it specifies how the block acts on the input state. The input state is either entry state or exist state, depending on whether it is a forward or backward analysis.

These two types of analysis are designed for accomplishing different analysis task. In this thesis, we will use forward analysis.

Transfer Function The transfer function can change the values of interest in an input state. It is defined by simulating the semantics of statements (in our case, bytecode instructions) in a block.

For example, the block labeled 111 is an INVOKEVIRTUAL instruction, which executes an out action designated at a location (with type LogicalLocality). Note that the actual value of this parameter location (userserver) is only known to the instruction after simulation of the execution of the block labelled 109 (ASM analysis framework takes care of this and we will explain it later). The transfer function of this block could be defined, such as:

- add out action to the *action set* that is associated with a certain instruction executed previously. Each action set collects actions which occur after a certain instruction;
- add userserver to the *destination location set* (for out action) that is associated with a previously executed instruction;
- add userserver to a *predicted data-flow set* associated with a value defined

in a previously executed instruction and indicates that userserver reaches this instruction.

Solving Analysis Equations For each block i, we shall also define a *join operation* ($join$) to combine the exit states (in case it is a forward analysis) of all predecessor blocks of i, in order to generate the entry state of block i. By using the transfer function of block i (f_i), the analysis equations are defined as follows:

$$\text{exit}_i = f_i\ (\text{entry}_i)$$
$$\text{entry}_i = join\ (\text{exit}_j), \text{ where exit}_j \text{ represents the predecessor nodes of } i$$

The join operation will also influence the definition of transfer function. For example, the transfer function of block 153-154 shall be defined in a similar way as block 109 which takes the exit state of previously executed instruction (block 152) as an input. Moreover, it also takes and incorporates the program facts from the exit state of its parameter process (in this case, from block labelled 158).

It would be relatively easy to solve these equations if AspectKlava did not support the eval action, because each entry block has unique predecessor exit block. We can just sort the edges in the control flow graph of a process, specify the initial analysis values to be empty at the entry of the first block, and compute the analysis values at the boundary of each block in a sorted (sequential) order. Afterwards we can extract analysis properties based on those analysis results (program facts) associated with each block.

We adapted a worklist algorithem [NNH05] to solve the equations when AspectKlava supports the eval action, in which case some of the entry blocks may have multiple predecessor blocks. We will discuss this shortly.

Flow-sensitive Analysis Note that the data-flow analysis we perform is *flow sensitive*, which means that the analysis results of a process with instructions ... instr_i; instr_j; ..., is different from the analysis results of a process with instructions ... instr_j; instr_i; ..., see the instructions come in a different order.

We need to perform flow sensitive analysis because the program facts include data-flow information of various variables among bytecode instructions (of a process). Program facts will change if the order of these instructions are changed.

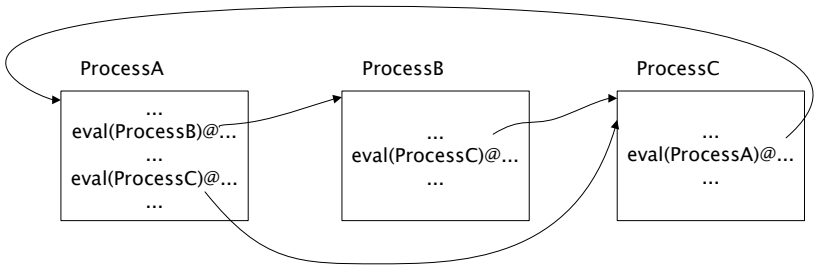

Figure 7.4: Call Graph of Processes

7.3.2.3 Inter-procedural Analysis and Worklist Algorithm

As the control flow in Figure 7.3 indicated, we have to obtain first the analysis results from the parameter process of an eval action, when computing analysis results (performing the transfer function) about eval action. This type of analysis is named *interprocedual analaysis*, because functions and procedures are taken into account (in our case, the functions and procedures are AspectKlava processes). On the contrary, process that not allows eval action can be analyzed by *intraprocedual analysis*, where functions or procedures are not involved. The intraprocedual fragment of our data-flow analysis has already been described in Section 7.3.2.2.

The definitions of AspectKlava processes can be mutually recursive. As shown in Figure 7.4, the analysis result of ProcessA is dependent on the analysis result of ProcessB and ProcessC, while the analysis result of ProcessB depends on the analysis result of ProcessC, which in turn depends on the results from ProcessA. The call relationship among these processes can be considered as constraints as below.

$$\begin{array}{lcl} \mathsf{ProcessA} & \sqsupseteq & \mathsf{ProcessB},\ \mathsf{ProcessC} \\ \mathsf{ProcessB} & \sqsupseteq & \mathsf{ProcessC} \\ \mathsf{ProcessC} & \sqsupseteq & \mathsf{ProcessA} \end{array}$$

Since it is impossible to go through each process definition just once and to compute all analysis results of these processes, we need to use an iterative algorithm. The algorithm computes a number of times the analysis results of these process definitions until the results remain stable for all the processes. In other words, if we consider each process has only one holistic transfer function operating on all its instructions, we shall compute the *least solution* of the constraints system, i.e., reach the so-called *least fixed point* lfp [NNH05] of each process's transfer function:

$$\mathrm{lfp}(f_i) = \sqcap \mathrm{Fix}(f_i), \text{where } \mathrm{Fix}(f_i) = \{l | f_i(l) = l\},$$

and $i \in \{\mathsf{ProcessA}, \mathsf{ProcessB}, \mathsf{ProcessC}\}$,
and l is an analysis result

An efficient way to compute the constraint system is through an worklist algorithm. The general idea is to use a *worklist* to control the iteration: the tasks to be done are stored in the worklist (in our case the tasks are to be solved constraints); the iteration selects a task from the worklist and removes it afterwards from the list. The processing of the task may cause new tasks to be added to the worklist. This process is iterated until there are no more tasks to be done, i.e., the worklist is empty. We will return to this algorithm later.

7.3.3 Using the ASM Framework for Bytecode Analysis

A number of bytecode translation libraries can generate, transform, and analyze compiled Java classes that are represented as byte arrays. In this thesis, we choose to use ASM [BLC02], a lightweight all purpose Java bytecode manipulation and analysis framework, to accomplish our analysis tasks. Other tools such as BCEL [Dah01], Javassist [Chi98] and Soot [VRCG+99] are able to accomplish the same tasks as well but with different complexity and runtime performance. For example, Soot provides a much more comprehensive analysis framework than ASM, which can be used to develop very advanced analysis. However, it requires to perform more expensive analyses for completing the same analysis task, thus it might sacrifice the runtime performance for relatively simple analysis tasks.

7.3.3.1 ASM Analysis Framework

ASM provides tools to read, write and transform the compiled Java class (byte arrays) by working on higher-level concepts of a class than bytes. The data-flow analysis framework of ASM relies on the object-based representation of a class, where a class is represented by a tree of objects, and each object represents a part of the class, such as a field, a method, etc.

In the Java Virtual Machine execution model, Java code is executed inside a *thread*, which has its own execution stack. Each time a method is invoked, a new *execution frame* is pushed to the current thread's execution stack. Each frame contains a *local variables* part and an *operand stack* part. The local variables part contains variables that are accessed by their index. The operand stack is a stack of values used by bytecode instructions.

The forward data-flow analysis in ASM is performed by simulating the execution of a method's bytecode instructions on an *abstract execution frame*. This

involves the simulation of popping values from and pushing values back to the frame's operand stack. If there exist any branch instructions, the execution of both branches will be simulated and the data-flow analysis shall be able to combine values from both branches.

The overall data-flow analysis algorithm for a single method, the task of popping from and pushing back the low-level abstract values to the stack is already implemented by the ASM analysis framework. Thus we focus on defining *low-level abstract values* for the basic data types in AspectKlava (which are used as the instruction's operands), the *high-level abstract values* for program facts, the transfer functions for each instruction that collect program facts, and on extending the single method analysis to interprocedural analysis in order to accomplish our analysis task.

7.3.3.2 Program Facts Computation and Integration

Analysis Domain and Transfer functions Both low-level abstract values and high-level abstract values can be considered as our analysis domain, because they are essentially both abstract data that approximate the properties of a program, but at a different level.

Low-level abstract values are popped from and pushed back when simulating the execution of a method's bytecode instructions. For example, when the analysis loads the bytecode version of class ALogicalLocation, an abstract value of this class is needed for it to be virtually executed by the ASM framework.

High-level abstract values are collected for answering our high level analysis questions. For each process class, the high-level abstract values map each action to its respective analysis result obtained from its continuation instructions. Note that low-level abstract values can also be used in high-level abstract values.

When defining the transfer function for each instruction *trans_inst*, we need to manipulate both low-level and high-level abstract values. The low-level abstract values are relatively easy to push around, as we only need to follow the semantics of each instruction; the high-level abstract values are more challenging to use, because for a given instruction, we have to update them based the property of interest. Defining a transfer function for the high-level abstract values are generally discussed in Section 7.3.2.2. For example, when encountering an out action, the high-level abstract values require to update action sets (associated with those previously executed actions), while this is not only based on the original semantics of the bytecode instructions, it also relies on our analysis interest. Finally, all the transfer functions of an instruction accumulated together can (or will) form a higher level transfer function, namely *trans_proc*.

Key Steps of the Interprocedural Data-flow Analysis To use a worklist algorithm to perform interprocedural analysis, we have to go through the following steps:

1. Constraints Collection: Start from the first process to be analyzed, we use a depth-first search (DFS) algorithm to traverse the call graph of processes, and collect all constraints that denote the dependency relationship between processes. Please refer to Section 7.3.2.3 for a constraints example. Note that we use the ASM analysis framework to traverse the method AexecuteProcess (of AspectKlava) in order to collect constraints.

2. Analysis Result Initialization: We initialize the analysis result (formed by abstract values) of all dependent processes as empty.

3. Analysis Results Computation: Start from the first process to be analyzed, and collect analysis results for the current process. We use ASM analysis framework to traverse the constructor method and AexecuteProcess method (of AspectKlava), and compute the analysis results from the entry state (formed by both low-level and high-level abstract values) of the process via using its transfer function *trans_proc* (which consists of several *trans_inst* for each instructions). Note that this will need the analysis results from other processes when there is eval action in this process.

4. Keep Updating Analysis Results until Stable: Use constraints to guide the iteration of the algorithm, and re-analyze a process only if its callee process (the parameter process in a eval action) modifies the analysis result. The analysis results will keep growing until it reaches the fixed point of transfer function *trans_proc*. Note that to ensure the algorithm terminates, analysis domain, transfer function and join function have to be defined in a particular way so that the analysis result either stays the same or grows larger, and the analysis result cannot grow indefinitely. Refer to [NNH05] for more details.

After the algorithm terminates, we obtain program facts (analysis result) for all processes reachable from the starting process, and they then will be cached for future use. The join point actions in these processes can be linked with their program facts, which will be used by program analysis predicates and functions at runtime.

Integration of Program Facts into Aspects To use program facts in aspects, we establish an internal link between program facts and aspects.

- By using *useID* (a pair of integers $\langle actionIx, paramIx \rangle$, introduced in Section 7.1.1) as both the index to structure program facts (consists of abstract values) and the index to locate join point actions and their parameters in the runtime system, each program fact can be mapped to a join point action (with *actionIx*), and moreover, it is also possible to select relevant parts of a corresponding program fact for a particular parameter of a matched join point (with *paramIx*).

- The data types used in program facts shall be converted to those used in aspects, even though these facts are originally collect from data type used in the main program. For example, in the AspectKlava process, a constant location is defined using class LogicalLocation. Our analyzer will first read and convert its bytecode format into a corresponding abstract value that is used in the analysis, and then returns a constant expressed in class ALogicalLocation, so that it can be recognized and used in aspects.

- Parameters defined in a pointcut can match and bind a runtime value from the join point as in other AOP programming language, but it needs to be annotated with appropriate *useID*, to link and compare with values from the program facts, in order to realize the so-called dual value evaluation mechanism.

Example 7.6 Table 7.2 displays the program facts (analysis results) obtained from the bytecode instructions shown in Listing 7.8. They are used in aspects to evaluate program analysis predicates and functions at runtime.

The first column shows that the program facts of a process are indexed by action number. Number 1-4 represent the *actionIx* within process clientsendmsg in Listing 6.5, e.g, number 1 denotes the in action. Number -1 represents the program fact at the beginning of a process, which is useful when computing program facts for other processes (that invoke this process such as clientlogin).

The second and third column show the analysis domains and analysis values (high-level abstract value), which are used at runtime in aspects. Note that the analysis values are represented with a mixed value of internal data type (low-level abstract value) used in ASM framework (e.g, L1, L2, etc), and aspect datatype (e.g., #AL_2 is a value of type ALogicalLocality with 2 as the *paramIx* number). Here we avoid explaining the meaning of these data in detail, as these program facts are used internally in the runtime system for evaluating program analysis predicates and functions.

Note that Table 7.2 only shows the program facts obtained from analyzing Listing 7.8 (with AspectKE* source code in Listing 6.5). If we start the analysis on bytecode instructions of Listing 6.4, we need also to analyze all the processes it spawns: clientsendmsg, clientreceivemsg, clientsendfile, and clientreceivefile. With

#Action (*actionIx*)	Analysis Domain	Analysis Values
-1	DefinedSet	[L1, L2, L3]
	ArgumentMap	{3=L3, 2=L2, 1=L1}
	ActionSet_X	[in, out, eval]
	UsedValueMap_X	{out={L1=[#AL_2], L2=[#AL_2, Eavesdropper]}, read={}, in={L1=[#AL_1], L2=[#AL_1]}, eval={L1=[#AL_1]}, newloc={}}
	LocationMap_X	{out=[#AL_2, Eavesdropper], read=[], in=[#AL_1], eval=[#AL_1], newloc=[]}
	ActionSet_Y	[]
	UsedValueMap_Y	{out={}, read={}, in={}, eval={}, newloc={}}
	LocationMap_Y	{out=[], read=[], in=[], eval=[], newloc=[]}
1	DefinedSet	[L1, L2, K1, L3, L4]
	ArgumentMap	{4=K1, 3=L4, 2=L2, 1=K2, 0=L1}
	ActionSet_X	[in, out, eval]
	UsedValueMap_X	{out={L1=[#AL_2], L2=[#AL_2, Eavesdropper], K1=[#AL_2], L4=[#AL_2, Eavesdropper]}, read={}, in={L1=[#AL_0], L2=[#AL_0]}, eval={L1=[#AL_0]}, newloc={}}
	LocationMap_X	{out=[#AL_2, Eavesdropper], read=[], in=[#AL_0], eval=[#AL_0], newloc=[]}
	ActionSet_Y	[]
	UsedValueMap_Y	{out={}, read={}, in={}, eval={}, newloc={}}
	LocationMap_Y	{out=[], read=[], in=[], eval=[], newloc=[]}
2	DefinedSet	[L1, L2, K1, L3, L4]
	ArgumentMap	{4=L1, 3=K1, 2=L4, 1=K3, 0=L2}
	ActionSet_X	[in, out, eval]
	UsedValueMap_X	{out={L1=[#AL_0], L2=[#AL_0, Eavesdropper], L4=[Eavesdropper]}, read={}, in={L1=[#AL_4], L2=[#AL_4]}, eval={L1=[#AL_4]}, newloc={}}
	LocationMap_X	{out=[#AL_0, Eavesdropper], read=[], in=[#AL_4], eval=[#AL_4], newloc=[]}
	ActionSet_Y	[]
	UsedValueMap_Y	{out={}, read={}, in={}, eval={}, newloc={}}
	LocationMap_Y	{out=[], read=[], in=[], eval=[], newloc=[]}
3	DefinedSet	[L1, L2, K1, L3, L4]
	ArgumentMap	{2=L4, 1=L2, 0=L5}
	ActionSet_X	[in, out, eval]
	UsedValueMap_X	{out={L1=[#AL_1], L2=[#AL_1, Eavesdropper]}, read={}, in={L1=[#AL_-3], L2=[#AL_-3]}, eval={L1=[#AL_-3]}, newloc={}}
	LocationMap_X	{out=[#AL_1, Eavesdropper], read=[], in=[#AL_-3], eval=[#AL_-3], newloc=[]}
	ActionSet_Y	[]
	UsedValueMap_Y	{out={}, read={}, in={}, eval={}, newloc={}}
	LocationMap_Y	{out=[], read=[], in=[], eval=[], newloc=[]}
4	DefinedSet	[L1, L2, K1, L3, L4]
	ArgumentMap	{3=L3, 2=L2, 1=L1, 0=L1}
	ActionSet_X	[]
	UsedValueMap_X	{out={}, read={}, in={}, eval={}, newloc={}}
	LocationMap_X	{out=[], read=[], in=[], eval=[], newloc=[]}
	ActionSet_Y	[in, out, eval]
	UsedValueMap_Y	{out={L1=[#AL_2], L2=[#AL_2, Eavesdropper]}, read={}, in={L1=[#AL_1], L2=[#AL_1]}, eval={L1=[#AL_1]}, newloc={}}
	LocationMap_Y	{out=[#AL_2, Eavesdropper], read=[], in=[#AL_1], eval=[#AL_1], newloc=[]}

Table 7.2: Program Facts of clientsendmsg

the worklist algorithm, program facts from all these processes will be computed, and cached for future use of aspect evaluation. □

7.4 Concluding Remarks

To summarize this chapter, we presented the detailed implementation techniques for AspectKlava runtime system.

Firstly we described the high level implementation strategy regarding static analysis: Gather fundamental static analysis information at process's load time, to avoid performing all dynamic syntax-based program analysis at runtime. Apparently, this strategy is more efficient than the semantics defined in AspectKE, and we will give benchmark results in the next chapter. Our innovative static-dynamic dual value evaluation mechanism enables a uniform way of expressing security conditions no matter whether it is checked statically or dynamically.

Secondly, we present the core programming elements used in AspectKlava, which further extends the Klava package with aspect-oriented notation.

Thirdly, we review the actual interprocedural data-flow analysis that has been developed. As the processes will be executed in a distributed mobile environment, where source code is not available, our analysis is performed on Java bytecode. We also describe how the program facts (analysis results) can be linked to aspect evaluation at runtime system.

Chapter 8

Demonstration and Evaluation

In the previous two chapters we have presented the design of the AspectKE* programming language and its runtime system AspectKlava. In this Chapter, we shall discuss the usability, performance, and expressiveness of this language. Section 8.1 starts by presenting simple demonstrations using AspectKE* to build a distributed chat system. We show how a security hole is fixed by a simple aspect, that takes the power of program analysis predicates and functions to detect malicious actions. Section 8.2 assesses the expressiveness and performance of our language via case studies and benchmarking.

8.1 Demonstration

In this section, we demonstrate with screenshots how AspectKE* can fix a security hole in the distributed chat system presented in Chapters 6 and 7. We assume readers understand the architecture of the chat system, the proposed security policies, and definitions of the relevant processes and aspects. All of them have been presented in Section 6.2.2.

Demo 1: Chat System without Malicious Code Figure 8.1 displays a screen shot of the distributed chat system which contains no malicious code. It shows that Alice using the left ChatConsole exchanges messages and files with

Bob who is using the right ChatConsole.

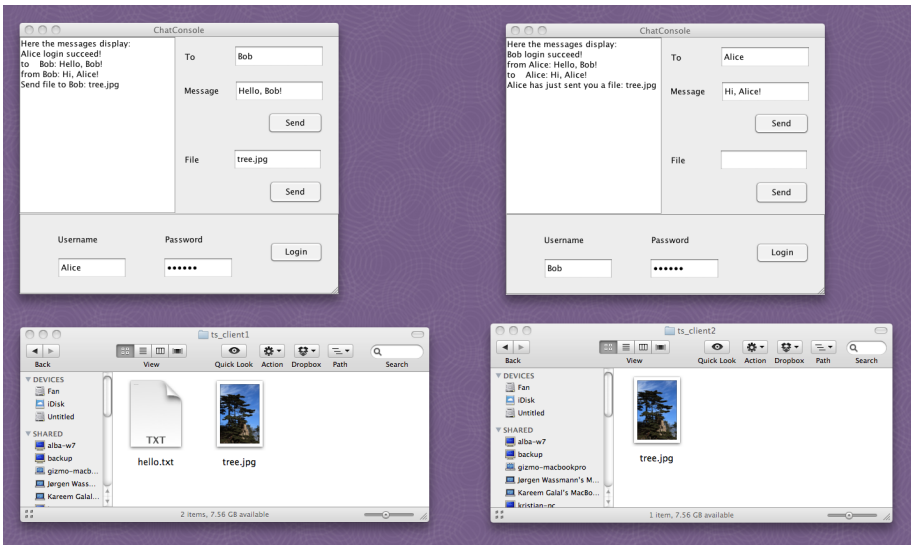

Figure 8.1: Chat System without Malicious Code

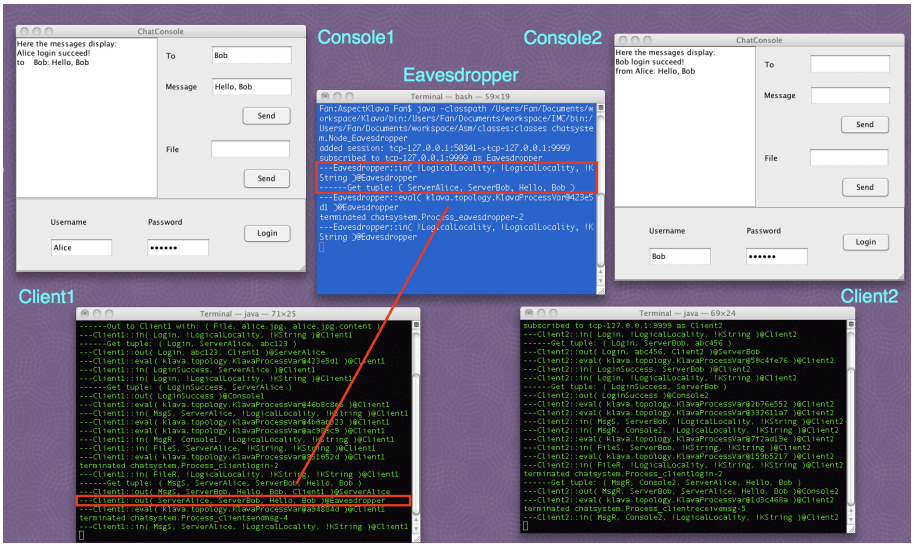

Figure 8.2: Information is Leaked from Client1 and Client2 to Eavesdropper

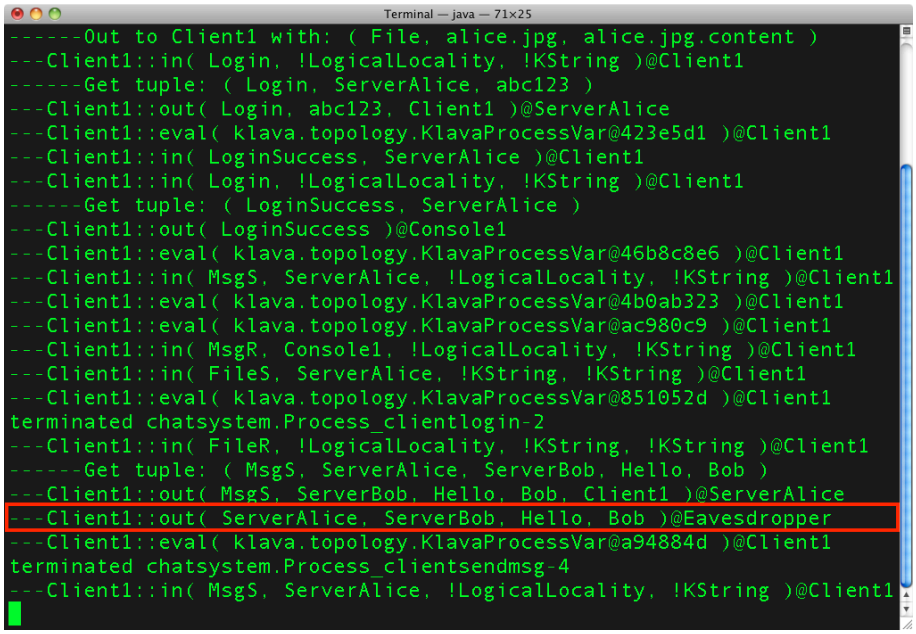

Figure 8.3: Client1 Sends a Message to Eavesdropper

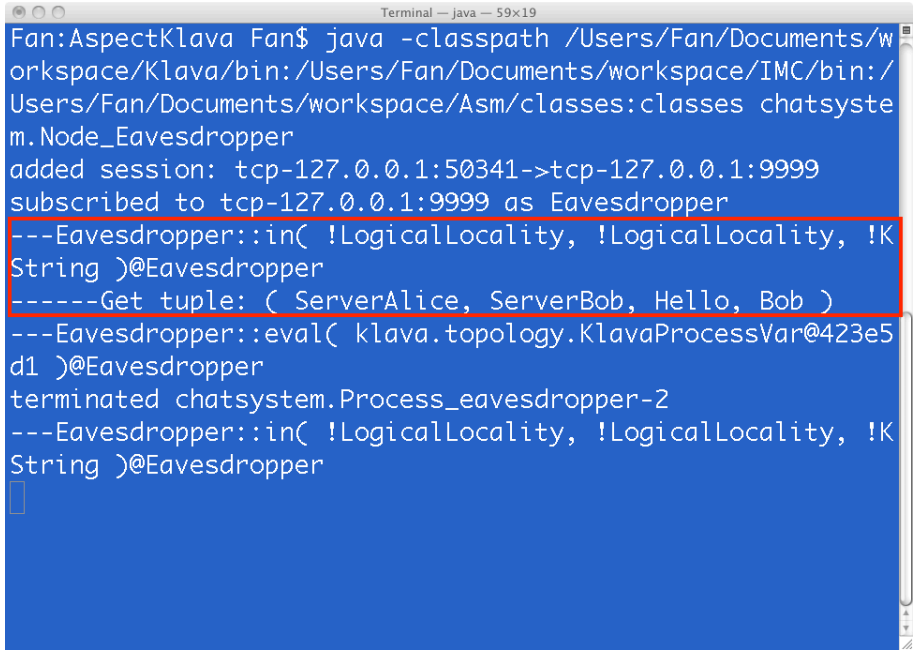

Figure 8.4: Eavesdropper Receives a Message Sent from Client1

Demo 2: Chat System with Security Hole In this demo, the chat system contains a security hole, which might be exploited by a malicious process and thus violates Policy 2 (presented in Section 6.2.2). Figure 8.2 shows the execution traces where a malicious client process runs at node Client1 and an eavesdropper process (Listing 8.1) runs at node Eavesdropper. In this demo, no malicious actions are performed at process clientlogin (removes Line 8 in Listing 6.4), but process clientsendmsg (Listing 6.5) executes malicious actions as follows (using the following code to replace Line 9 in Listing 6.5).

```
out( userserver , friendserver , text )@Eavesdropper;
```

Figures 8.3 and 8.4 zoom in on the execution traces of processes executed at Client1 and Eavesdropper. In the orange boxes, we can see that the messages captured by the malicious clients are received by the Eavesdropper node. Clearly, the security hole has been exploited successfully by the untrusted process.

```
1 proc eavesdropper(){
2   location  userserver , friendserver ;
3   string  text ;
4
5   in( userserver , friendserver , text )@Eavesdropper;
6   eval( process  eavesdropper() )@Eavesdropper;
7 }
```

Listing 8.1: Process eavesdropper

Demo 3: Chat System with Security Hole Removed In this demo, the chat system contains a security hole as in Demo 2, but the hole is removed by an aspect, which prevents malicious process from exploiting the hole. Figure 8.5 shows the execution traces where a security aspect is enforced (defined in Listing 6.7). Figure 8.6 zooms in on the execution traces of processes executed at Client1. In the orange box, we can see that the aspect terminated client process clientsendmsg because it detected that there are malicious actions in the continuation process and security policy might be violated. Clearly, the security hole is removed by the aspect.

From the above demonstrations we can see how an aspect (that reasons about the future behavior of a process) can terminate the execution of a process that contains malicious actions to be executed in future. It shows how simple it is to specify aspects to fix security holes in a distributed system by using AspectKE*.

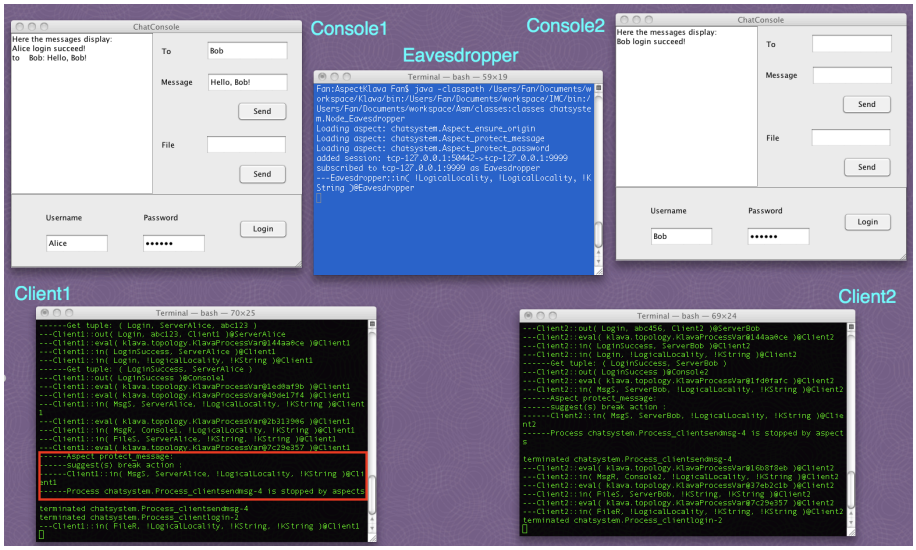

Figure 8.5: No information is Leaked: Client1 (and Client2) are Terminated by Aspect protect_message

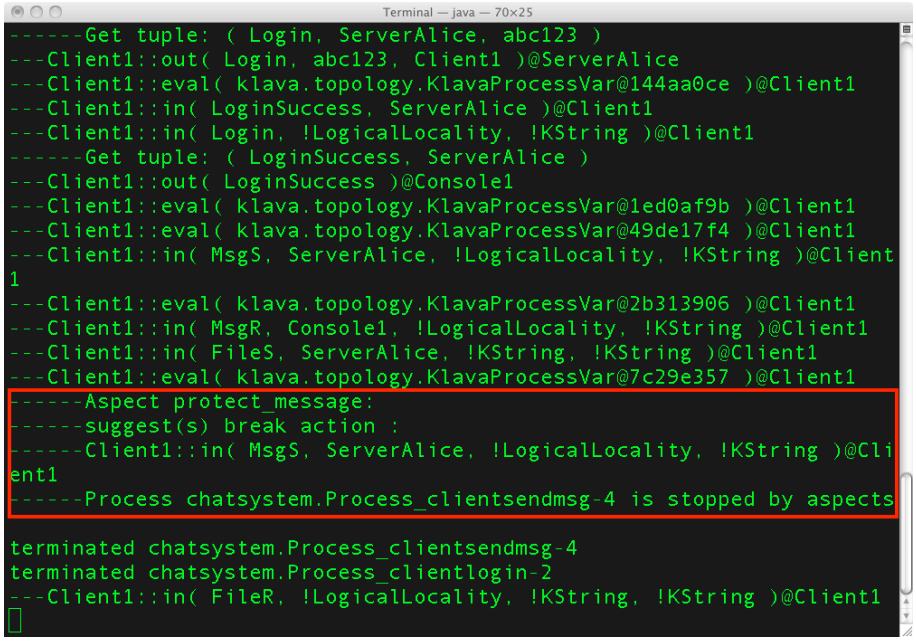

Figure 8.6: Client1 is Terminated by Aspect protect_message

8.2 Evaluation

In this section, we assess the expressiveness and performance of our language through case studies and benchmarking.

8.2.1 Performance Evaluation

Our performance evaluation focuses on the overheads of program analysis rather than the entire execution time of practical application programs. This is because our implementation is merely a prototype, whose execution time could be largely different from a fully-optimized implementation. The purpose of performance evaluation is to validate our implementation framework.

We compared overheads of basic operations in "our implementation" with that in a "naive implementation". Our implementation analyzes a program only once to gather program facts at load-time for later usage at runtime. The naive implementation analyzes a program whenever a program analysis predicate or function is to be evaluated. The latter simply simulates the semantic model of AspectKE, as we already presented in Chapters 3 and 4.

case	configuration			our implementation			naive implementation			time improvement (%)
	process	#executions	#aspects	#analysis performed	time of load-time analysis	time of process execution	#analysis performed	time of runtime analysis	time of process execution	
1	clientlogin	1	0	1	34	28	0	0	31	
	clientsendmsg	10	0	0	0	205	0	0	190	
	Total			1	267		0	221		−21
2	clientlogin	1	1	1	37	34	1	37	33	
	clientsendmsg	10	1	0	0	199	10	145	199	
	Total			1	270		11	414		35
3	clientlogin	1	1	1	37	32	1	36	36	
	clientsendmsg	100	1	0	0	1966	100	1443	1847	
	Total			1	2035		101	3362		39
4	clientlogin	1	1	1	35	40	1	36	34	
	clientsendmsg	10	10	0	0	265	100	1340	271	
	Total			1	340		101	1681		80

Table 8.1: Benchmark Results of Chat System (msec.)

We use a program that sequentially executes processes clientlogin and clientsendmsg (presented in Section 6.2) in the chat system, with aspects protect_password and protect_message enforced. Compared with definitions shown in Section 6.2, the definitions here are slightly simplified to avoid termination of the processes and inclusion of other processes into analysis.

The compiled (bytecode) version of process clientlogin contains 113 instructions, and clientsendmsg contains 89 instructions. Our analyzer directly performs static analysis on them. The programs were executed on top of JVM 1.6.0 for MacOS X by a 2.16 GHz Intel Core 2 Duo processor with 2 GB memory. The time is measured by System.currentTimeMillis(). We execute each program for 100 times and calculated the median values.

Table 8.1 summarizes the measurement results. We executed the benchmark program with four configurations of different message and aspect numbers. For each implementation, the table lists the number of program analyses performed, as well as the time spent for analysis and process execution. Note that even though process execution time includes time for aspects activation and evaluation, it excludes the time spent for performing (load-time/runtime) static analysis, which is listed in separate columns.

From the table, we can see that our implementation analyzes programs even when no aspects are applied (case 1). The analysis count for clientsendmsg is zero in our implementation, because the interprocedual analysis performed on clientlogin analyzes clientsendmsg in advance. While the time a naive implementation spends for analysis is proportional to message and aspect numbers, our implementation spends constant time for analysis.

As shown in the rightmost column of case 1, our implementation has overheads (21% in this measurement) due to load-time program analysis even if no aspects are applied. The overheads are quickly compensated for when there are aspects.

Although it is difficult to predict performance of a fully optimized implementations, we can clearly claim, from this measurement, that huge overheads of static program analysis make naive implementations far from realistic. We can also presume that our implementation strategy can give realistic performance to programs that require less dynamic process migrations but have many aspects.

8.2.2 Comparison Against Analysis-Based AOP Languages

We argue that our approach offers better abstraction than existing analysis-based AOP languages. In particular, policies that require both runtime and static information cannot be easily implemented by others.

Some advanced AOP languages [AM07,CN04,KRH04] allow programmers to define their own pointcut primitives, including those that exploit program analysis results. In theory, it is possible for those languages to define security aspects based on future behavior of a program, via defining pointcuts that statically analyze the program. However, those languages offer access to programs at bytecode or AST-level, which makes it difficult to correctly implement static

analyses. For example, Josh [CN04] provides an extensible pointcut language, where analysis-based pointcuts can be defined based on the Javaassist library. As Javaassist does not directly provide data and control flow of a program, users have to develop the analysis almost from scratch. SCoPE [AM07] is an extended AspectJ compiler that allows programmers to define analysis-based pointcuts which specify join point shadows by using user-defined static program analysis, and provides access to various bytecode manipulation libraries, including the ones offering powerful program analysis capabilities (e.g., Soot [VRCG+99]). However, users still have to develop the analysis at low-level. Moreover, these languages do not provide a mechanism to combine runtime data and static information as we do.

Below, we show an aspect for Policy 2 (the policy is defined in Section 6.2) in SCoPE (Policy 1 in SCoPE is as simple as in AspectKE* because it does not depend on any static program analysis).

```
1  class ChatClient{
2    ChatServer getMyServer(){...}
3    void sendMessage(ChatServer to, String message){
4      getMyServer().sendMessage(this,to,message);
5    }
6     ...
7  }
8  class ChatServer{
9    ChatClient getMyClient(){...}
10   void sendMessage(ChatClient sender, ChatServer to,
11     String message){
12     to.sendMessage(this,message);
13   }
14    ...
15 }
```

Listing 8.2: Chat Server and Client (Simplified Version)

The base chat program (originally introduced in Section 6.2) for SCoPE aspects is re-implemented using Java Remote Method Invocation API. Listing 8.2 shows the fragment code of classes for the chat servers and clients[1]. A client cannot send a message directly to another client. Instead, it sends messages to the server associated to it using sendMessage in ChatServer. The associated server is obtained from getMyServer() in class ChatClient. The server then sends the message to another server which is specified by the second parameter of sendMessage in ChatServer, and finally the other client gets the message from its server through receiveMessage(not shown here) in ChatClient.

Listing 8.3 shows aspect ViolateP2, which terminates a program when Policy 2 is violated. The leakMessage() method checks whether sendMessage() is called on a ChatServer object who does not come from getMyServer(). It exploits

[1]These classes essentially extends Remote interface, here we present the simplified version.

```
1  pointcut violateP2():
2    withincode(* Console.*(..)) &&
3    call(void ChatClient.sendMessage(..)) &&
4    if(leakMessage( thisJoinPointStaticPart ));
5  boolean leakMessage(JoinPoint.StaticPart sjp){
6    InvkInst invk = getMethodInsnOf(sjp);
7    Collection<Method> impl = dispatch(sjp);
8    for(Method m:impl){
9      DataflowGraph dg = getDataflowGraph(m);
10     Value ths = impl.getThis();
11     Value msg = impl.getArg()[1];
12     Collection<InvkInst> methodInvokes = getInvocationsWithin(m);
13     Collection<Value> mySvrs = /* create a collection */
14     for(InvkInst node : methodInvokes)
15       if(node invokes ChatClient.getMyServer())
16         mySvrs.add(node.getReturn());
17       else if(node invokes ChatServer.sendMessage())
18         if(!dg.isOneOf(node.getTarget(),mySvrs))
19           return true;
20       else
21         for(Value arg: node.getArgs())
22           if(dg.flowTo(msg,arg))
23             return true;
24   }
25   return false;
26 }
27 before(): violateP2(){
28   terminate();
29 }
```

Listing 8.3: Aspect for Policy 2 in SCoPE

the methods that are commonly available in program analysis libraries (such as Soot [VRCG+99]), in order to get an intra-procedural data flow graph, a method invocation instruction which corresponds to the given join point shadow, a set of methods to which a method invocation instruction dispatches, and a set of method invocations within a method. The aspect is longer and more complicated than the one in AspectKE*, because SCoPE lacks abstractions for common tasks. For instance, the function targeted in AspectKE* successfully abstracts the locations to which a message will be sent, whereas a number of operations (12–18) need to be implemented in ViolateP2.

Moreover, this aspect does not fully implement Policy 2 because it has no access to runtime data. AspectKE*, however, can uniformly check it as primarily discussed in Section 6.3.3 and 7.1.2.

8.2.3 Security Policies for an Electronic Healthcare Record Workflow System

To assess expressiveness and applicability of AspectKE* for enforcing more realistic security policies, we implemented security policies for an electronic health-

care record workflow system. Some interesting steps in this system have already been modeled in Chapters 2.2, 3, 4 when presenting AspectKE. This study is to confirm that the policies modeled in Chapter 3 and 4 can be defined by exploiting language primitives of AspectKE*, especially the proposed program analysis predicates and functions.

The target system manages *electronic health record*s (EHRs) – a database that stores patients' data, and a place where doctors, nurses, administrators, researchers, etc perform different tasks relying on patients' EHR records. Note that most of the target system and policies are directly extracted from a health information system for an aged care facility in New South Wales, Australia [EB04]. Since policies can be determined internally by the system owner, or externally by laws, regulations, norms etc, we also integrate interesting security policies from the literature such as [Dep03b, Dep03a, Bez98, SBH+07, Can02]. Some are extracted from nationwide EHR security policies issued by governments, and others are from selected research papers. The aim is to cover both primary use of data (producing or acquiring data at an individual patient's care process) and emerging use of data (e.g., for purposes like public health and commercial activities). From another perspective, the policies cover both classical access control and predictive access control models.

We implemented a simple tuple space based EHR workflow system in AspectKE*. It consists of 16 nodes, 41 processes and all security policies (aspects) presented in Chapters 3 and 4, except for aspect $\mathsf{A}_{\mathsf{p7}}^{eval}$, as the behavior analysis functions LC and LC_c defined in Table 4.4 are currently not supported by AspectKE*. We plan to provide them in the future.

When using other AOP languages that support no analysis-based pointcuts (e.g., AspectJ), policies that depend on the classical access control models (presented in Chapter 3) can still be implemented, however policies that refer to predictive access control (presented in Chapter 4) are difficult to be implemented because they rely on future behavior of an action. (AOP languages with analysis-based pointcuts are discussed in Section 8.2.2.) When using other security mechanisms for tuple space systems, such as the ones based on Java Security framework [FAH99] or other techniques [VBO03, HR03, GLZ06], we could implement those policies in Chapter 3. However, policies presented in Chapter 4, cannot be implemented because those mechanisms do not provide information on future behavior.

In summary, our experience shows that AspectKE* is very expressive and useful to enforce complex real world security policies to a distributed system written in the KLAIM computing model.

8.3 Concluding Remarks and Related Work

In this chapter we have demonstrated the usability of AspectKE* by presenting three demos with a series of screenshots. The aim is to show how easy a security hole in a distributed chat system can be removed through an analysis-based aspect, which prevents a malicious process from exploiting it. Then the performance results were given, which testifies the two-stage implementation strategy introduced in previous chapters is more practical and more efficient than the original semantics of AspectKE. We also compared our work with other analysis-based AOP languages by building the same chat system, and highlight our advantages. In the end, we described our experience on enforcing various security policies to an electronic health care workflow system in AspectKE*.

As we have finished presenting both AspectKE and AspectKE*, we will discuss about the related work.

8.3.1 Expressive Pointcuts

AspectKE* provides high-level predicates and functions to describe future behavior of a program, which makes it much easier to implement security aspects than other analysis-based approaches [AM07, CN04, KRH04] since users do not need to develop the analysis. In particular, these high-level language constructs are designed with security properties in mind (can be considered as security policy language constructs), which is particularly useful for users to compose security policies, as users prefer to compose and enforce security policies with domain specific policy languages [DDLS00]. Recently, the `maybeShared()` pointcut (matches all field accesses that may be shared) was proposed in [BH10] to help implement efficient monitoring algorithms for detecting data race in concurrent systems, the aim – providing easily accessible pointcut for users to avoid developing complex static analysis – is similar to ours but in a different application domain.

Alpha [OMB05] provides Prolog queries to exploit information over a rich set of program semantics for the execution history up to the current join point. The language is very expressive and we believe it may even support predicates that combine static and runtime information of a variable as we did. However, it cannot capture the future events and is not a compiled language and therefore lacks efficient implementation.

In sum, these languages [AM07, CN04, KRH04] implement partially the security policies with more complicated definitions as we compared in Section 8.2.2, or they provide only sophisticated constructs without realistic implementation [OMB05]. AspectKE* can be considered as an approach to provide highly ex-

pressive pointcuts to AOP languages, like `pcflow` [Kic03], `maybeShared` [BH10], and the ones for distributed computing [NCT04,NSV+06,TT06]. However, none of the others are directly comparable to ours with respect to security policy enforcement to distributed applications.

8.3.2 Control- and Data-flows Pointcuts

There are several AOP systems in which a pointcut can specify a relationship between join points. AspectJ's `cflow` captures join points based on a control flow, which is useful to implement access control policies. `dflow` [MK03] identifies joint points based on data-flow information, which is useful to enforce secrecy and integrity policies. However, both capture control data flow that has already occurred, rather than that will happen in the future as in AspectKE*.

`pcflow` [Kic03] is proposed to reason about which join points can occur in the future under a program's control flow. `Tracematches` [AAC+05] and `transcut` [SMH09] are similar approaches for identifying temporal relationships between joint points, which describe related events by patterns comprised of several pointcuts. Tracematches track join point relationships based on execution traces, while transcut tracks relationships by selecting a region in the control flow graph of a program. However, none of the above-mentioned methods ever integrate with any data-flow analysis as AspectKE* does.

8.3.3 Access Control with Aspect-oriented Programming

There are plenty of studies for applying AOP to enforce access control policies. To name a few: [WJP02,CW09,dOWKK07]. To the best of our knowledge, only AspectKE* supports predictive access control policies. Moreover, as far as we know, we are the first to enforce secondary use of data policies through security aspects.

8.3.4 Tuple Space Security

There are tuple space systems which provide security mechanisms. For example, KLAIM [DFPV00] has a static type system that realizes access control. SecOS [VBO03] provides a low-level security mechanism that protects every tuple field with a lock. Secure Lime [HR03] provides a password-based access control mechanism for building secure tuple spaces in ad hoc settings. JavaSpaces [FAH99], which is used in industrial contexts, has a security mechanism based on the Java security framework. Our work is different in using AOP with

program analysis. Hence it not only provides a flexible way to enforce security policies, but also enables predictive access control policies and secondary use of data policies, which cannot be realized in these other approaches.

Chapter 9

Conclusion

In this dissertation, we have investigated the use of Static Program Analysis techniques in conjunction with the Aspect-oriented Programming approach to model and build distributed, mobile systems, using tuple spaces. Our main thesis is that

> Aspect-oriented programming provides a flexible way of enforcing security policies in distributed systems, more specifically, within the tuple space paradigm. Static program analysis techniques can enhance the expressiveness of security aspects and elegantly support the enforcement of security policies that rely on information flow.

We have presented AspectKE, AspectK and AspectKE*, three aspect-oriented extension of the coordination language KLAIM [BDNFP98], which provide concrete vehicles for presenting our approach. The distributed tuple spaces provide a natural model of the kind of system that motivated our work. However, the approach could equally well be applied to other distributed frameworks, especially those based on more classical process calculi. The join points in this case would be read and write accesses to channels (e.g, π-calculus).

The static program analysis techniques integrated in the aspects are primarily based on data-flow analysis, which is demonstrated to be particularly useful for enforcing security policies that require explicit information flow like predictive

access control and secondary use of data when integrating within aspects. However, the approach could also be combined with other static program analysis techniques and thus enforce other type of security policies, which we shall briefly discuss in the end of the thesis.

The remaining parts of this chapter will recapitulate the technical contributions we developed and discuss a number of directions for future work.

9.1 Contributions

- We have specified AspectKE, an aspect-oriented extension of the coordination process calculus KLAIM, equipped with formal semantics which describes how aspect-oriented notation is introduced into the tuple space paradigm by trapping actions (Chapter 3), and how behavior analyses are developed and integrated into aspects (Chapter 4). To illustrate the latter idea, we allow the pointcuts to trap processes, either continuation processes or processes to be executed remotely, and introduce several simple *behavior analysis functions* for collecting information from the trapped processes, and use them in advice.

- We have discussed some language extensions building on top of the AspectKE model and proposed the notion of *open joinpoint* which universally exists in coordination languages such as tuple space system. These are primarily discussed in the way of formalizing the process calculus AspectK (Chapter 5), which allows actions *before* and *after* **proceed** or **break** in an advice.

- We evaluated the expressiveness of AspectKE (and AspectK) by investigating its policy enforcement capability through examples in an electronic health care (EHR) setting. We have demonstrated that AspectKE can enforce discretionary access control, mandatory access control as well as role-based access control. Furthermore, AspectKE can elegantly retrofit new policies to an existing system with minimum effort at any phase within the system development cycle. We have also shown that in a distributed and mobile system, the information flow is very hard to control. AspectKE can enforce a range of *predictive access control policies* to cater for this issue, through composing aspects that check remote evaluation and the program continuation. We enforce both *primary and secondary use of data policies*, which shows that AspectKE is suitable to cater for old as well as new challenges in a complex distributed computing environment, where security and privacy are of great importance. We have also shown that AspectK can benefit crosscutting activities such as logging as the advices do in other AOP languages. These are primarily covered in Chapters 3, 4 and 5, along with the presentation of our process calculi.

- We have designed and implemented a proof-of-concept programming language AspectKE*, based on AspectKE programming model developed in previous chapters. One of the key features of AspectKE* is the *program analysis predicates and functions* that provide information on future behavior of a program. With a *dual value evaluation mechanism* that handles results of static analysis and runtime values at the same time, those functions and predicates enable programmers to specify security policies in a uniform manner. Our *two staged implementation strategy* gathers fundamental static analysis information at load-time, to avoid performing all analysis at runtime as AspectKE does. We built a compiler for AspectKE* and developed a runtime system – AspectKlava in Java. This includes the development of interprocedural data-flow analysis on processes in the form of bytecode instructions. The main materials are presented in Chapters 6 and 7.

- We have demonstrated the usability of AspectKE*, assessed expressiveness and the performance of implementation primarily through examples in a distributed chat applications setting. Besides implementing a secure distributed chat application (Chapters 6 and 8), we also successfully implemented the security policies presented in Chapter 3 and 4, and enforced them to an electronic health care workflow system using AspectKE*.

  To conclude, we find that the combination of aspects with behavior and static analysis techniques shows great potential for serving as a flexible and powerful mechanism for policy enforcement, and a promising method for building security and trust in a distributed and mobile environment.

9.2 Future Work

Below we outline several directions for future work relevant to the AspectKE/AspectKE* programming model.

9.2.1 Indirect Flow Analysis

There are other challenging secondary use of data policies which not only require control of *direct flows* but also of *indirect flows* [DD77,SM03]: e.g. after storing specific data into a doctor's own tuple space, the doctor should not allow them to be transferred into a researcher's tuple space by indirectly passing through another location. In this case, checking all the parallel executing processes with security aspects that rely on more advanced static analyses might be needed. For example, the static analysis should include analysis components that are able to

predict all possible data that can be stored in a certain tuple space. It could be done using the flow logic framework [HPN06, HNNP08]), or developing analysis techniques by combining pointer analysis [NNH05] for tuple spaces with control and data-flow analysis over processes. In this way, we shall also use aspect to express policies depending on indirect flows.

9.2.2 Enriching the Power of Aspects

The current AspectKE/AspectKE* language can only make the monitored process terminate or proceed. It would be interesting to extend the language so that it can perform other kinds of actions. In AspectKE/AspectKE* we disallow before and after actions around the **proceed**/**break** advice (as allowed in AspectK). This is because, if we had allowed these actions, a safe behavior analysis would be very difficult to achieve, since the processes *to be executed* might execute more actions (inserted by aspects at runtime) than expected. This is an interesting direction for future work and will require more powerful program analyses than the analyses this dissertation presented, as we need to incorporate effects from aspects while analyzing processes.

9.2.3 Formal Validation of AspectKE Programs

As AspectKE has a formal semantics, we shall be able to formally formulate a static analysis and verify whether a given AspectKE program (including both base and aspect programs) satisfies certain global security properties that are expressed by the analysis result. In this way, we shall know whether the workflow system holds certain global security properties after enforcing the complete set of EHR policies presented in this dissertation (Chapters 3 and 4).

Bibliography

[A+72] J. Anderson *et al.*, "Computer Security Technology Planning Study." 1972.

[AAC+05] C. Allan, P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA'05*. ACM, 2005, p. 364.

[AGMO06] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, "An overview of CaesarJ," in *Transactions on Aspect-Oriented Software Development I*, ser. LNCS, vol. 3880. Springer, 2006, pp. 135–173.

[AM07] T. Aotani and H. Masuhara, "SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts," in *Proceedings of the 6th international conference on Aspect-oriented software development, AOSD'07*. ACM, 2007, pp. 161–172.

[And00] J. G. Anderson, "Security of the distributed electronic patient record: a case-based approach to identifying policy issues," *International Journal of Medical Informatics*, vol. 60, no. 2, pp. 111–118, 2000.

[And01] J. H. Andrews, "Process-algebraic foundations of aspect-oriented programming," in *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting*

Concerns, REFLECTION'01, ser. LNCS, vol. 2192. Springer, 2001, pp. 187–209.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[Bar84] H. Barendregt, *The lambda calculus: its syntax and semantics*. North Holland, 1984.

[Bar92] H. P. Barendregt, "Lambda calculi with types," pp. 117–309, 1992.

[BBC+06] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, p. 85, 2006.

[BBD+03] L. Bettini, V. Bono, R. De Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri, "The klaim project: Theory and practice," in *Proceedings of the Global Computing. Programming Environments, Languages, Security, and Analysis of Systems, IST/FET International Workshop, GC'03*, ser. LNCS, vol. 2874. Springer, 2003, pp. 88–150.

[BDNF+04] L. Bettini, R. De Nicola, D. Falassi, M. Lacoste, L. Lopes, L. Oliveira, H. Paulino, and V. T. Vasconcelos, "A Software Framework for Rapid Prototyping of Run-Time Systems for Mobile Calculi," in *Global Computing. IST/FET International Workshop, GC 2004, Revised Papers*, ser. LNCS, C. Priami, Ed., vol. 3267. Springer, 2004, pp. 179–207.

[BDNFP98] L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese, "Interactive Mobile Agents in X-Klaim," in *Proceedings of the 7th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE'98*. IEEE Computer Society, 1998, pp. 110–115.

[BDP02] L. Bettini, R. De Nicola, and R. Pugliese, "Klava: a Java package for distributed and mobile applications," *Software-Practice and Experience*, vol. 32, no. 14, pp. 1365–1394, 2002.

[Bez98] K. Beznosov, "Requirements for access control: US healthcare domain," in *Proceedings of the third ACM workshop on Role-based access control, RBAC'98*. ACM, 1998, p. 43.

[BFIK99] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The keynote trust-management system version 2," 1999, RFC 2704.

[BH10] E. Bodden and K. Havelund, "Aspect-oriented Race Detection in Java," *IEEE Transactions on Software Engineering*, 2010.

[BJJR04] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely, "μabc: A minimal aspect calculus," in *Proceedings of the 15th International Conference on Concurrency Theory, CONCUR'04*, ser. LNCS, vol. 3170. Springer, 2004, pp. 209–224.

[BK08] C. Baier and J. Katoen, *Principles of model checking.* The MIT Press, 2008.

[BLC02] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems," in *Proceedings of the ASF (ACM SIGOPS France) Journees Composants 2002: Adaptable and extensible component systems*, 2002.

[BLW05] L. Bauer, J. Ligatti, and D. Walker, "Composing security policies with polymer," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI'05.* ACM, 2005, pp. 305–314.

[BO05] P. A. Bonatti and D. Olmedilla, "Driving and monitoring provisional trust negotiation with metapolicies," in *Proceedings of the 6th IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY'05.* IEEE Computer Society, 2005, pp. 14–23.

[BS04] M. Y. Becker and P. Sewell, "Cassandra: Flexible trust management, applied to electronic health records," in *Proceedings of the 17th IEEE Computer Security Foundations Workshop, CSFW'04.* IEEE Computer Society, 2004, pp. 139–154.

[Can02] Canadian Institutes of Health Research, *Secondary Use of Personal Information in Health Research: Case Studies.* Public Works and Government Services Canada, 2002, located at http://www.cihr-irsc.gc.ca/e/1475.html.

[CCBR06] F. Cuppens, N. Cuppens-Boulahia, and T. Ramard, "Availability enforcement by obligations and aspects identification," in *Proceedings of the The 1st International Conference on Availability, Reliability and Security, ARES'06.* IEEE Computer Society, 2006, pp. 229–239.

[CH05] A. W. Colman and J. Han, "Coordination systems in role-based adaptive software," in *Proceedings of the 7th International Conference on Coordination Models and Languages, COORDINATION'05*, ser. LNCS, vol. 3454. Springer, 2005, pp. 63–78.

[Che05] F. Chen, "Java-MOP: A monitoring oriented programming environment for Java," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05*, ser. LNCS, vol. 3440. Springer, 2005, pp. 546–550.

[Chi98] S. Chiba, "Javassist-a reflection-based programming wizard for Java," in *Proceedings of OOPSLAâ98 Workshop on Reflective Programming in C++ and Java*, 1998.

[CM04] A. Charfi and M. Mezini, "Aspect-oriented web service composition with AO4BPEL," in *Proceedings of the European Conference on Web Services, ECOWS'04*, ser. LNCS, vol. 3250. Springer, 2004, pp. 168–182.

[CN04] S. Chiba and K. Nakagawa, "Josh: an open AspectJ-like language," in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD'04*. ACM, 2004, pp. 102–111.

[CSZ04] S. Capizzi, R. Solmi, and G. Zavattaro, "From endogenous to exogenous coordination using aspect-oriented programming," in *Proceedings of the 6th International Conference on Coordination Models and Languages, COORDINATION'04*, ser. LNCS, vol. 2949. Springer, 2004, pp. 105–118.

[CW09] B. Cannon and E. Wohlstadter, "Enforcing security for desktop clients using authority aspects," in *Proceedings of the 8th ACM international conference on Aspect-oriented software development, AOSD'09*. ACM, 2009, pp. 255–266.

[Dah01] M. Dahm, "Byte code engineering with the BCEL API," Freie Universitat Berlin, Technical Report, 2001.

[Dan00] Danish Data Protection Agency, *The Act on Processing of Personal Data*, 2000, located at http://www.datatilsynet.dk/english/the-act-on-processing-of-personal-data/.

[Dan07] D. S. Dantas, "Analyzing security advice in functional aspect-oriented programming languages," Princeton University, Phd Dissertation, 2007.

[DD77] D. Denning and P. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[DDLS00] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "A language for specifying security and management policies for distributed systems," London: Department of Computing, Imperial College, Tech. Rep., 2000.

[Dep03a] Department of Health, UK, *Integrated Care Records service: Output Based Specification version 2*, 2003, located at http://www.dh.gov.uk/en/Publicationsandstatistics/Publications/PublicationsPolicyAndGuidance/DH_4118312.

[Dep03b] Department of Health, UK, *NHS Code of Practice-Confidentiality*, 2003, located at http://www.dh.gov.uk/en/Publicationsandstatistics/Publications/PublicationsPolicyAndGuidance/DH_4069253.

[DeT02] J. DeTreville, "Binder, a logic-based security language," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy, SP'02*. IEEE Computer Society, 2002, pp. 105–113.

[DFP98] R. De Nicola, G. L. Ferrari, and R. Pugliese, "KLAIM: A kernel language for agents interaction and mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 315–330, 1998.

[DFP00] R. De Nicola, G. L. Ferrari, and R. Pugliese, "Programming access control: The KLAIM experience," in *Proceedings of the 11th International Conference on Concurrency Theory, CONCUR'00*, ser. LNCS, vol. 1877. Springer, 2000, pp. 48–65.

[DFPV00] R. De Nicola, G. L. Ferrari, R. Pugliese, and B. Venneri, "Types for access control," *Theoretical Computer Science*, vol. 240, no. 1, pp. 215–254, 2000.

[DGH+08] R. De Nicola, D. Gorla, R. R. Hansen, F. Nielson, H. R. Nielson, C. W. Probst, and R. Pugliese, "From flow logic to static type systems for coordination languages," in *Proceedings of the 10th International Conference on Coordination Models and Languages, COORDINATION'08*, ser. LNCS, vol. 5052. Springer, 2008, pp. 100–116.

[DHS07] C. Duma, A. Herzog, and N. Shahmehri, "Privacy in the semantic web: What policy languages have to offer," in *Proceedings of the 8th IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY'07*. IEEE Computer Society, 2007, pp. 109–118.

[dOWKK07] A. S. de Oliveira, E. K. Wang, C. Kirchner, and H. Kirchner, "Weaving rewrite-based access control policies," in *Proceedings of*

the 2007 ACM workshop on Formal methods in security engineering, FMSE'07. ACM, 2007, pp. 71–80.

[EB04] M. Evered and S. Bögeholz, "A case study in access control requirements for a health information system," in *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation, ACSW Frontiers'04*. Australian Computer Society, Inc., 2004, pp. 53–61.

[ES00] Ú. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy, SP'00*. IEEE Computer Society, 2000, pp. 246–255.

[ET99] D. Evans and A. Twyman, "Flexible policy-directed code safety," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy, SP'99*. IEEE Computer Society, 1999, pp. 32–45.

[FAH99] E. Freeman, K. Arnold, and S. Hupfer, *JavaSpaces principles, patterns, and practice*. Addison-Wesley, 1999.

[FF05] R. E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Aspect-Oriented Software Development*. Boston: Addison-Wesley, 2005, pp. 21–35.

[FLZ06] R. Focardi, R. Lucchi, and G. Zavattaro, "Secure shared dataspace coordination languages: a process algebraic survey," *Science of Computer Programming*, vol. 63, no. 1, pp. 3–15, 2006.

[GDY+04] 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 the 16th International Conference on Software Engineering & Knowledge Engineering, SEKE'04*, 2004, pp. 360–365.

[Gel85] D. Gelernter, "Generative communication in Linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.

[GLZ06] R. Gorrieri, R. Lucchi, and G. Zavattaro, "Supporting secure coordination in SecSpaces," *Fundamenta Informaticae*, vol. 73, no. 4, pp. 479–506, 2006.

[Gol99] D. Gollmann, *Computer Security*, 1st ed. John Wiley & Sons, 1999.

[GRF02] G. Georg, I. Ray, and R. B. France, "Using aspects to design a secure system," in *Proceedings of the 8th International Conference on Engineering of Complex Computer Systems, ICECCS'02)*. IEEE Computer Society, 2002, pp. 117 – 126.

[Ham06] K. W. Hamlen, "Security policy enforcement by automated program-rewriting," Cornell University, Phd Dissertation, 2006.

[HJ08] K. W. Hamlen and M. Jones, "Aspect-oriented in-lined reference monitors," in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security, PLAS'08*. ACM, 2008, pp. 11–20.

[HNN09] C. Hankin, F. Nielson, and H. R. Nielson, "Advice from belnap policies," in *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF'09*. IEEE Computer Society, 2009, pp. 234–247.

[HNNP08] R. R. Hansen, F. Nielson, H. R. Nielson, and C. W. Probst, "Static validation of licence conformance policies," in *Proceedings of the The Third International Conference on Availability, Reliability and Security, ARES'08*. IEEE Computer Society, 2008, pp. 1104–1111.

[HNNY08] C. Hankin, F. Nielson, H. R. Nielson, and F. Yang, "Advice for coordination," in *Proceedings of the 10th International Conference on Coordination Models and Languages, COORDINATION'08*, ser. LNCS, vol. 5052. Springer, 2008, pp. 153–168.

[Hoa78] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, p. 677, 1978.

[HPN06] R. R. Hansen, C. W. Probst, and F. Nielson, "Sandboxing in myKlaim," in *Proceedings of the The First International Conference on Availability, Reliability and Security, ARES'06*. IEEE Computer Society, 2006, pp. 174–181.

[HR03] R. Handorean and G. Roman, "Secure sharing of tuple spaces in ad hoc settings," *Electronic Notes in Theoretical Computer Science*, vol. 85, no. 3, pp. 122–141, 2003.

[JJR03] R. Jagadeesan, A. Jeffrey, and J. Riely, "A calculus of untyped aspect-oriented programs," in *Proceedings of the 17th European Conference on Object-Oriented Programming, ECOOP'03*, ser. LNCS, vol. 2743. Springer, 2003, pp. 54–73.

[KFJ03] L. Kagal, T. W. Finin, and A. Joshi, "A policy based approach to security for the semantic web," in *Proceedings of the Second International Semantic Web Conference, ISWC'03*, ser. LNCS, vol. 2870. Springer, 2003, pp. 402–418.

[KHH+01] 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*, ser. LNCS, vol. 2072. Springer, 2001, pp. 327–353.

[Kic03] G. Kiczales, "The fun has just begun," *Keynote AOSD*, 2003.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming, ECOOP'97*. Springer, 1997, pp. 220–242.

[KRH04] G. Kniesel, T. Rho, and S. Hanenberg, "Evolvable pattern implementations need generic aspects," in *Proceedings of the Workshop on Reflection, AOP, and Meta-Data for Software Evolution, RAM-SE'04*. Fakultät für Informatik, Universität Magdeburg, 2004, pp. 111–126.

[LCX+01] T. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman, "Hitting the distributed computing sweet spot with TSpaces," *Computer Networks*, vol. 35, no. 4, pp. 457–472, 2001.

[Ler01] X. Leroy, "Java bytecode verification: an overview," in *Proceedings of the 13th International Conference on Computer Aided Verification, CAV'01*, ser. LNCS, vol. 2102. Springer, 2001, pp. 265–285.

[LJ06] B. Lagaisse and W. Joosen, "True and transparent distributed composition of aspect-components," in *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware, Middleware'06*. Springer, 2006, pp. 42–61.

[LY99] T. Lindholm and F. Yellin, *Java(TM) Virtual Machine Specification*. Addison-Wesley Professional, 1999.

[Mil82] R. Milner, *A calculus of communicating systems*. Springer, 1982.

[MK03] H. Masuhara and K. Kawauchi, "Dataflow pointcut in aspect-oriented programming," in *Proceedings of the 1st Asian Symposium on Programming Languages and Systems, APLAS'03*, ser. LNCS, vol. 2895. Springer, 2003, pp. 105–121.

[MKD03] H. Masuhara, G. Kiczales, and C. Dutchyn, "A compilation and optimization model for aspect-oriented programs," in *Proceedings of the 12th International Conference on Compiler Construction, CC'03s*, ser. LNCS, vol. 2622. Springer, 2003, pp. 46–60.

[MPW92] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, i," *Information and computation*, vol. 100, no. 1, pp. 1–40, 1992.

[MZZ+01] A. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java information flow," *Software release.*, 2001, located at http://www.cs.cornell.edu/jif.

[Nat03] National Board of Health, *National IT Strategy 2003-2007 for the Danish Health Care Service.* The Ministry of the Interior and Health, 2003.

[NCT04] M. Nishizawa, S. Chiba, and M. Tatsubori, "Remote pointcut: a language construct for distributed AOP," in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD'04.* ACM, 2004, pp. 7–15.

[Nec97] G. C. Necula, "Proof-carrying code," in *POPL'97.* ACM, 1997, pp. 106–119.

[NN02] H. R. Nielson and F. Nielson, "Flow Logic: a multi-paradigmatic approach to static analysis," *The essence of computation*, pp. 223–244, 2002.

[NN07] H. R. Nielson and F. Nielson, *Semantics with applications: an appetizer.* Springer, 2007.

[NNH05] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*, 2nd ed. Berlin, Germany: Springer, 2005.

[NSV+06] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. D. Fraine, and D. Suvée, "Explicitly distributed AOP using AWED," in *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD'06.* ACM, 2006, pp. 51–62.

[Oak98] S. Oaks, "Java security," *O'Reilly & Associates, Inc. Sebastopol, CA, USA*, p. 456, 1998.

[OMB05] K. Ostermann, M. Mezini, and C. Bockisch, "Expressive pointcuts for increased modularity," in *Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05*, ser. LNCS, vol. 3586. Springer, 2005, pp. 214–240.

[OT00] H. Ossher and P. Tarr, "Hyper/J: multi-dimensional separation of concerns for Java," in *Proceedings of the 22nd international conference on Software engineering, ICSE'00*. ACM, 2000, pp. 734–737.

[PGA02] A. Popovici, T. Gross, and G. Alonso, "Dynamic weaving for aspect-oriented programming," in *Proceedings of the 1st international conference on Aspect-oriented software development, AOSD'02*. ACM, 2002, pp. 141–147.

[Plo81] G. Plotkin, "A structural approach to operational semantics. Report DAIMI FN-19," *Computer Science Department, Aarhus University*, 1981.

[PS08] P. H. Phung and D. Sands, "Security policy enforcement in the OSGi framework using aspect-oriented programming," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC'08*. IEEE Computer Society, 2008, pp. 1076–1082.

[PSD+04] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli, "JAC: an aspect-based distributed dynamic framework," *Software Practice and Experience*, vol. 34, no. 12, pp. 1119–1148, 2004.

[QZW+85] L. Qiu, Y. Zhang, F. Wang, M. Kyung, and H. Mahajan, "Trusted computer system evaluation criteria," *National Computer Security Center*, 1985.

[RPW06] R. Ramachandran, D. J. Pearce, and I. Welch, "AspectJ for multilevel security," in *Proceedings of the 5th Workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS'06*. University of Virginia, 2006, pp. 1–5.

[SBH+07] C. Safran, M. Bloomrosen, W. Hammond, S. Labkoff, S. Markel-Fox, P. Tang, and D. Detmer, "Toward a National Framework for the Secondary Use of Health Data: An American Medical Informatics Association White Paper," *Journal of the American Medical Informatics Association*, vol. 14, no. 1, pp. 1–9, 2007.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[Sch00] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 1, pp. 30–50, 2000.

[SM03] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5– 19, 2003.

[SMH01] F. B. Schneider, J. G. Morrisett, and R. Harper, "A language-based approach to security," in *Informatics - 10 Years Back. 10 Years Ahead*, ser. LNCS, vol. 2000. Springer, 2001, pp. 86–101.

[SMH09] H. Sadat-Mohtasham and H. Hoover, "Transactional pointcuts: designation reification and advice of interrelated join points," in *Proceedings of the 8th international conference on Generative programming and component engineering, GPCE'09*. ACM, 2009, pp. 35–44.

[TJSJ08] E. Truyen, N. Janssens, F. Sanen, and W. Joosen, "Support for distributed adaptations in aspect-oriented middleware," in *Proceedings of the 7th international conference on Aspect-oriented software development, AOSD'08*. ACM, 2008, pp. 120–131.

[TT06] E. Tanter and R. Toledo, "A versatile kernel for distributed aop," in *In Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006*, ser. LNCS, vol. 4025. Springer, 2006, pp. 316–331.

[UES00] Úlfar Erlingsson and F. B. Schneider, "SASI enforcement of security policies: a retrospective," in *Proceedings of the workshop on New security paradigms, NSPW'99*. ACM, 2000, pp. 87–95.

[VBO03] J. Vitek, C. Bryce, and M. Oriol, "Coordinating processes with secure spaces," *Science of Computer Programming*, vol. 46, no. 1-2, pp. 163–193, 2003.

[VPDW+05] T. Verhanneman, F. Piessens, B. De Win, E. Truyen, and W. Joosen, "Implementing a modular access control service to support application-specific policies in caesarj," in *Proceedings of the 1st workshop on Aspect oriented middleware development, AOMD'05*. ACM, 2005.

[VRCG+99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[Wan01] M. Wand, "A semantics for advice and dynamic join points in aspect-oriented programming," in *Proceedings of the 2nd International Workshop on Semantics, Applications, and Implementation of Program Generation, SAIG'01*, ser. LNCS, vol. 2196. Springer, 2001, pp. 45–46.

[Win05] K. T. Win, "A review of security of electronic health records," *Electronic Health Records: security, safety and archiving*, vol. 34, no. 1, pp. 13–18, 2005.

[WJP02] B. D. Win, W. Joosen, and F. Piessens, "Developing secure applications through aspect-oriented programming," in *Aspect-Oriented Software Development.* Addison-Wesley, 2002, pp. 633–650.

[WVD01] B. D. Win, B. Vanhaute, and B. D. Decker, "Security through aspect-oriented programming," in *Proceedings of the IFIP TC11 WG11.4 First Annual Working Conference on Network Security.* Kluwer, 2001, pp. 125–138.

[WZL03] D. Walker, S. Zdancewic, and J. Ligatti, "A theory of aspects," in *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming, ICFP'03.* ACM, 2003, pp. 127–139.

[YAM+] F. Yang, T. Aotani, H. Masuhara, F. Nielson, and H. R. Nielson, "AspectKE*:Aspects with High-level and Efficient Program Analysis for Securing Distributed Systems," submitted to a conference.

[YHNN] F. Yang, C. Hankin, F. Nielson, and H. R. Nielson, "Aspect-oriented access control of tuple spaces," submitted to a journal.

[YMA+10a] F. Yang, H. Masuhara, T. Aotani, F. Nielson, and H. R. Nielson, "AspectKE*:security aspects with program analysis for distributed systems," in *Proceedings of the 9th Workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS'10.* University of Potsdam, 2010, pp. 27–31.

[YMA+10b] F. Yang, H. Masuhara, T. Aotani, F. Nielson, and H. R. Nielson, "AspectKE*:security aspects with program analysis for distributed systems," in *Demonstration track of the 9th International Conference on Aspect-Oriented Software Development, AOSD'10*, 2010.