A Framework for Security Assurance of Access Control Enforcement

Jaime A. Pavlich-Mariscal
Departamento de Ingeniería de Sistemas y Computacion.
Universidad Católica del Norte.
Angamos 0610. Antofagasta. Chile.
jpavlich@ucn.cl

Steven A. Demurjian
Department of Computer Science & Engineering.
The University of Connecticut.
Unit-2155, 371 Fairfield Road, Storrs, CT 06269-2155, USA.
steve@engr.uconn.edu

Laurent D. Michel
Department of Computer Science & Engineering.
The University of Connecticut.
Unit-2155, 371 Fairfield Road, Storrs, CT 06269-2155, USA.
ldm@engr.uconn.edu

March 3, 2010

Abstract

Modeling of access control policies, along with their implementation in code, must be an integral part of the software development process, to ensure that the proper level of security in an application is attained. Previous work of the authors in this area yielded a framework that incorporates access control at the design and code levels, through a set of new extensions to UML and a set of approaches to enforce access control in an application [28]. An essential property of the code that has not been addressed by that framework is security assurance, which, in the context of this research, is to insure that the application code behaves consistently with the access control policy. This paper proposes a security
assurance mechanism that formalizes the application behavior using labeled transition systems and structural operational semantics [30]. Simulation relations [23] are used to demonstrate the correctness of the access control code with respect to the design. To validate the approach, this paper proves correctness of two access control enforcement mechanisms that are part of our case study: a basic approach to implement access control in code and an aspect-oriented approach.

1 Introduction

Access control is defined as: “Limiting access to information system resources only to authorized users, programs, processes or other systems” [36]. In today’s world, access control is an essential component to ensure that applications’ information is secure, uncorrupted, and available. The incorporation of access control into software presents an important challenge: most access control requirements are often discovered after functional requirements are defined and implemented [9]. As a result, access control flaws are not found and corrected at an early stage. Access control concerns added at latter stages in the software process, particularly post-implementation, can increase security defects and their cost of repair [22]. Therefore, it is very important that access control becomes a first-class concern of the software development process, specially at earlier stages in the software life-cycle. Overall, the general problem that motivates this research is: the need of a process for secure software engineering that incorporates access control at every stage in the software development process.

Our previous work in this area [28] consists of a framework to model access control policies and realize them into code (see Figure 1). At the design level, the focus is to separate the main design (non-access-control concerns) from the access control design. To assist designers to visualize access control policies, the framework includes a set of access control diagrams, i.e., extensions to the UML to model access control. To better adapt to changing access control requirements, each access control diagram comprises a set of access control features, i.e., composable units that realize specific capabilities of access control models, namely RBAC [14], MAC [6] and DAC [11]. Designers can select and compose features to achieve the desired behavior in an access control policy. The access control design maps into the access control code, which constrains the behavior of the application at runtime, based on the access control policy. The framework includes different approaches to translate access control models to code that preserve separation of access control concerns from the design.

At the code level, a very important issue is security assurance. In the context of our research (access control), security assurance means to insure that the application precisely realizes all of its access control
requirements. In practice, this requires the inclusion of appropriate mechanisms that enforce, at runtime, the policy specified by the access control model. The approach of manually coding access control enforcement mechanisms is risky, since programmers can make mistakes when realizing access control from design models. Although automatic code generation can assist developers to incorporate access control into the application, it is not sufficient to provide security assurance at the code level. To insure that the application code has no errors that could potentially lead to access control breaches, it is crucial to prove that the enforcement code correctly implements the access control from the design. A correct realization of the access control design means that the application behaves exactly as the policy intends, allowing subjects to access operations and objects only if allowed by the rules in the access control design. To perform such a proof, a formal model of the application at the design and code levels is essential, which details the way access control enforcement affects the behavior of the application.

This paper proposes an approach to provide security assurance through a proof of correctness of the enforcement code, and validates the approach applying this proof to different enforcement mechanisms that are part of a case study. Section 2 describes a software system utilized as a case study. The experience obtained during its development yielded some of the essential ideas for this paper. Section 3 details the access control code facets of the case study: the main assumptions of the proof of correctness regarding access control and two strategies to enforce access control in the case study application. Section 4 details the proof of correctness and discusses the scope of this kind of proofs. Section 5 validates the approach, proving that the two strategies for access control code correctly implement an access control design. Section 6 discusses related work. Section 7 concludes.
## 2 The University Application Case Study

The essential ideas of our previous work and this paper are based on the experience of the first author in the development of a university application. This application was the courseware system utilized by the Universidad Católica del Norte (“Northern Catholic University”, located in Antofagasta, Chile) from 2003 to 2007. The access control policy of the system comprises 10 roles, assigned to approximately of 6000 users per semester, and 122 permissions. Roles are organized in a hierarchy that determines the access to course materials. In addition, course owners (teachers) can delegate functions to assistants or students, and they can grant or deny people to access their courses or groups within courses. Similar policies apply to forums, workgroups, syllabi, and wikis within the application.

To better explain the main concepts of our work, this paper utilizes a simpler example based on the original university application. Figure 2 is a class model of the simplified application. **CourseDescription** manages all of the course information that is independent of time, i.e., course numbers, syllabi, prerequisites, etc. **CourseSection** manages the information of each course section per term, i.e., enrolled students, teachers, etc. **StudentInformation** manages information about students. **Catalog** manages the publicly-available information on courses offered at a university. **Logger** records events in the system.

![Class Model of the University Application Example](image)

**Figure 2:** Class Model of the University Application Example.

## 3 Access Control Code

This section further details the access control facets of the university application case study, which are essential to understand and validate the proposed approach for security assurance.
The access control approach in this paper assumes that the application code has the structure of Figure 3. The main code realizes the main concern of the application. For example, in the university application, the main code comprises all of the code that implements the methods of CourseSection, StudentInformation, Catalog, and CourseDescription. The public interface is the portion of the main code that is available to subjects who interact with the application. In the university application, the public interface comprises all of the public methods of the aforementioned classes. Not all of the methods in the public interface require protection from external access. For instance, according to the requirements of the university application, the methods from Catalog are publicly accessible, thus they do not require access control. The subset of the public interface that requires access control is the secure subsystem.

In practice, subjects do not directly access the methods in the public interface. For example, in non-distributed applications, the GUI code may be the intermediary between subjects and the public interface. Applications in a distributed architecture may use middleware code to access the public interface [26,35]. In general, there is code that is outside of the main code that accesses the public interface on behalf of the subjects. This paper calls that code the external code. The structure of the external code is assumed to be irrelevant, since it only performs calls to methods in the public interface; the focus is on the main code and the access control code. Therefore, the external code will not be further detailed.

The access control code (see Figure 3) has three main components. The policy code stores the access control policy (which methods and class instances of the secure subsystem are authorized to each subject). The access control enforcement code protects the public interface from any calls made from the external code and intercepts every such call to check whether it is allowed/denied according to the policy code and deny access to the method if necessary. The session code manages the interaction between the subjects and the system.

A very important assumption for the access control code is that the supporting infrastructure (programming language, execution environment, operating system, hardware, etc.) is adequately protected against intrusions from malicious users. Our approach for security assurance focuses on insuring consistency between the access control code and design models. Therefore, other concerns are outside the scope of this paper.

The remainder of this section details the enforcement code and its relation with the session and policy code. Section 3.1 describes a basic approach for access control enforcement. Section 3.2 describes an approach that utilizes aspect-oriented programming to enforce access control.
3.1 The Basic Enforcement

This section describes the simplest approach for access control enforcement that was utilized during the first iterations of the development of the university application. The basic approach consists of incorporating the enforcement code at all of the methods of the secure subsystem. The enforcement code is inserted at the beginning of each method, and controls access to that method based on subject privileges.

To illustrate the basic enforcement, Figure 4 is the code of CourseDescription before incorporating access control and Figure 5 shows the same class after incorporating security to setCourseNumber and setSyllabus (for space reasons, the remaining methods are not shown). These methods with access control allow the execution of the original code only if the active subject (the subject interacting with the application) is authorized to access each method, otherwise an exception is raised.

3.2 The Aspect-Oriented Enforcement

This section describes the aspect-oriented enforcement, which utilizes aspect-oriented programming (AOP) [20] to provide access control. AOP provides a modular unit, the aspect, to isolate crosscutting concerns, which are concerns that tend to be spread across the entire application and tangled with the implementation of other concerns. An aspect has two elements: the advices, which are portions of code that realize a crosscutting concern, and the pointcuts, which are the specifications of the places where the crosscutting concern code must be inserted. An aspect weaver, which is a compiler extension, weaves the code of the advices at the places specified by the pointcuts. Since access control is a crosscutting concern [8], AOP is a good candidate to realize access control enforcement. Figure 6 is the AccessControl aspect.
public class CourseDescription {
    boolean courseNumberModified=false;
    boolean syllabusModified=false;
    Integer courseNumber;
    String syllabus;
    Set<CourseDescription> prerequisites = new HashSet<CourseDescription>();
    public Integer getCourseNumber() {
        return courseNumber;
    }
    public Set<CourseDescription> getPrerequisites() {
        return prerequisites;
    }
    public String getSyllabus() {
        return syllabus;
    }
    public void setCourseNumber(Integer courseNumber) {
        this.courseNumber = courseNumber;
        courseNumberModified=true;
    }
    public void setSyllabus(String syllabus) {
        this.syllabus = syllabus;
        syllabusModified=true;
    }
}

Figure 4: The CourseDescription Class without Access Control.

public class CourseDescription {
...
    public void setCourseNumber(Integer courseNumber) {
        if (Session.isAuthorized("setCourseNumber",this,courseNumber)) {
            this.courseNumber = courseNumber;
            courseNumberModified=true;
        } else {
            throw new RuntimeException("Access Denied");
        }
    }
    public void setSyllabus(String syllabus) {
        if (Session.isAuthorized("setSyllabus",this,syllabus)) {
            this.syllabus = syllabus;
            syllabusModified=true;
        } else {
            throw new RuntimeException("Access Denied");
        }
    }
}

Figure 5: The CourseDescription Class with Basic Enforcement.
that modularizes all of the access control checks. The advice is woven before every method in the secure subsystem to check whether each method is authorized or not, according to the policy. If the subject is not authorized, the code raises a runtime exception, and denies access to the method and the object.

4 The Proof of Correctness

The essential idea that underlies the proposed approach for security assurance is that, to insure that the application code has no errors that could potentially lead to security breaches, it is crucial to prove that the enforcement code correctly implements the access control from the design. A correct implementation of the design means that the behavior of the code is consistent with the design specification. In terms of access control, a correct implementation of the access control design means that the application code behaves exactly as the policy intends, allowing subjects to access operations and objects only if allowed by the rules in the access control design. The proof of correctness, described in this section, formally insures the consistency between the application design and the code that realizes that design.

Figure 7 is an overview of the proof of correctness for security assurance. In the first iterations of the development of the software, the assumption is that the application has no access control; there exist an application design without access control and an application code without access control that realize all of the non-access-control requirements. A very important assumption is that the application code without access control correctly implements the design without access control. The reason to make this assumption is that there already exist mappings from standard UML diagrams into code that insure the correctness of
the implementation [16,19,21] for non-access-control concerns. Therefore, it is reasonable to assume that, before the incorporation of access control, software engineers use these mappings to generate code that is correct with respect to the design. When incorporating access control into the software, additional model elements are included in the design. The figure depicts this new design as the *application design with access control* that realizes access control and non-access-control concerns. Similarly, when incorporating access control into code, there must be code that enforces the access control policy at every method that needs protection. This code is called the *application code with access control*. To provide security assurance, one must prove that the application code with access control is a correct implementation of the application design with access control.

As described in our previous work [28], the application design without access control utilizes the Unified Modeling Language (UML) [25] to model the non-access control concerns the application. UML Class diagrams depict the application *structure*, represented as method signatures and class hierarchies. For example, Figure 2 (see Section 2) is a class diagram of the university application that specifies all of the classes and their methods. UML state diagrams describe the application *behavior*, the way the application state changes when each method is executed. For example, Figure 8 is a (very simplified) state diagram of the *CourseDescription* class from Figure 2. Each state, depicted as a rounded rectangle, represents changes in syllabus, course number, or both in an instance of *CourseDescription*. Transitions between states, depicted as arrows, are triggered by the call to methods *setSyllabus* or *setCourseNumber*.

At the code level, class declarations and method headers specify the structure of the application. The method bodies realize the behavior. For example, methods *setSyllabus* and *setCourseNumber* of *CourseDescription* (see Figure 4) change attributes *courseNumberModified* and *syllabusModified* to indicate that course number and syllabus have been modified, respectively.
To correctly implement the design, the code must reflect the behavior of the state diagrams of the application. For instance, the code of Figure 4 represents the states of Figure 8 with the attributes `courseNumberModified` and `syllabusModified`. When both attributes are false, the system is in the Unmodified state. When invoking `setCourseNumber`, the attribute `courseNumberModified` changes to true, which maps to the state Course number modified. If, thereafter, `setSyllabus` is invoked, the attribute `syllabusModified` changes to true, which maps to the state Course number and syllabus modified. Note that `CourseDescription` has attributes `courseNumber` and `syllabus` that map to additional states in the state diagram. For simplicity, this example omitted all of the states corresponding to these two attributes.

When incorporating access control into the application, the proof of Figure 7 must take into account the changes in the behavior of the application to implement an access control policy. For example, Figure 8 describes the behavior of `CourseDescription` without access control. There are no access control checks that could deny any of the transitions in this diagram. To realize an access control policy, the state diagrams of the design must change to allow only those transitions triggered by methods that are authorized according to the policy. For example, if a subject is authorized to access method `setSyllabus`, but is not allowed to access `setCourseNumber`, the state diagram of Figure 8 must change to the diagram of Figure 9, which does not allow the transitions that modify course numbers, transitioning instead into an exception state. The exception state means that the application denies the subject to access an operation and throws an access control exception that is handled by the system (e.g., displaying an “access denied” message) before continuing the execution of the application. The code that realizes a secure application must reflect the changes in the state diagrams, denying the execution of all of the methods forbidden.
by the policy, and executing only those methods that are authorized. For example, Figure 5 shows class \texttt{CourseDescription} with access control. Methods \texttt{setSyllabus} and \texttt{setCourseNumber} contain code that enforces permissions. The call to method \texttt{isAuthorized} returns true/false if the subject who interacts with the application is authorized/denied to access a method. The execution of each method continues only if \texttt{isAuthorized} returns true, otherwise an exception is raised.

To prove that the code correctly implements a design (with or without access control), one must have a mechanism to compare the behavior of the code with the behavior specified at the design level. To perform this comparison, both the code and the design must be represented using the same formalism. UML state diagrams are formalized as \textit{labeled transition systems} (LTS) [37], which are state transition systems where the transitions are labeled with the methods and objects that subjects access while interacting with the application. The code specification uses imperative \(\lambda\)-calculus, a small imperative language, whose semantics are formalized using \textit{structural operational semantics} (SOS) [30]. SOS provides a set of rules to evaluate expressions in a programming language that yield a transition system, where each state is represented as the values of all of the variables of the program that are in the memory of a computer and each transition is the execution of a method that changes the values of these variables. As Section 4.3 demonstrates, an SOS specification can be directly transformed into an LTS. This section uses \textit{simulation relations} [23] to demonstrate that the behavior of the application code, as specified by the derived LTS correctly implements the behavior of the application design, as specified by the LTS that represents UML state diagrams.

Section 4.1 details a formal model for access control policies. Section 4.2 formalizes the structure and behavior of an application at the design level as an LTS. Section 4.3 formalizes the way an application
behavior changes to enforce an access control policy by restricting all of the non-authorized transitions. Section 4.4 uses SOS to describe an imperative language to formalize the application code. Section 4.5 describes the way the application code implements an application design. Section 4.6 describes the conditions that the application code must satisfy to correctly implement a design. Section 4.7 discusses the scope of a proof of correctness, i.e., the level of detail to represent code elements in the proof.

4.1 Access Control Policy

This section describes the formal model for the design of applications and access control policies, as defined in our previous work [28]. This model is essential for the proofs of correctness.

The definition of an access control policy involves several key elements, depicted in Figure 10. There is a set of subjects, individuals who interact with the application. An application has two main elements: a set of operations available to subjects, called the public interface, and a set of objects that can be modified by those operations. The secure subsystem is the subset of operations in the public interface that need access control. The access control policy restricts the interaction between subjects, operations and objects.

Definitions 1 to 6 formalize the above elements.

Definition 1. Subj is the set of subjects that interact with an application.

Definition 2. PI is the public interface of an application, i.e., the set of operations available to subjects

Definition 3. Obj is the set of objects within the application.

Definition 4. SecSubs ⊆ PI is the set of operations in the public interface that require access control.
Definition 5. An authorization is a tuple \((s, op, obj)\), where \(s \in \text{Subj}\) is a subject that interacts with the application, \(op \in \text{SecSubs}\) is an operation of the secure subsystem, and \(obj \in \text{Obj}\) is an object of the application.

Definition 6. An access control policy \(P \subseteq \text{Subj} \times \text{SecSubs} \times \text{Obj}\) is a set of authorizations.

4.2 Application Design

The formal model of Section 4.1 describes the structure of the design of an application. To complete the formalization of the design, it is necessary to detail the behavior of the application: the way the internal state of the application changes when subjects execute operations over objects. UML state diagrams, which are used to model the behavior of an application are formalized in terms of labeled transition systems (LTS). Each state of an LTS comprises all of the objects of the application at each point in time, while transitions correspond to the execution of an operation of the public interface over an object. Formally, a labeled transition system is as follows:

Definition 7. A labeled transition system (LTS) is a tuple \((Q, PI, Obj, T)\), where \(Q\) is a set of states, \(PI\) is the set of operations available to subjects (the public interface); \(Obj\) is the set of objects of the application, and \(T \subseteq Q \times PI \times Obj \times Q\) is a transition relation that represents the change in the state of the application when a subject invokes an operation over an object.

Sometimes, \(T(q, op, obj) = q'\) is used to denote \((q, op, obj, q') \in T\), i.e., the transition from state \(q\) into \(q'\) when a subject invokes \(op\) over \(obj\).

An application, then, reduces to the coupling of a structure and a behavior (LTS), which is formalized below as:

Definition 8. An application design is a tuple \(APP = (PI, Obj, B)\), where \(PI\) is a set of operations available to subjects (the public interface), \(Obj\) is the set of objects of the application, and \(B = (Q, PI, Obj, T)\) is an LTS.

4.3 Application Design with Access Control

The incorporation of access control in the design consists of modifying an application without access control to restrict its behavior consistently with the access control policy. The execution of an operation of the public interface can occur only if authorized by the access control policy. This means that the LTS of the design must forbid all of those transitions triggered by prohibited operations. Let \(APP = (PI, Obj, B)\) be
the application design without access control and \( \text{as} \in \text{Subj} \) be the *active subject* that interacts with the application. To fold a policy \( P \subseteq \text{Subj} \times \text{SecSubs} \times \text{Obj} \) in the application design, \( \text{APP} \) must change into a new design \( \text{APP}^P = \langle \text{PI}, \text{Obj}, B^P \rangle \), where the active subject \( \text{as} \) can perform operations over objects, only if authorized by \( P \). The new LTS is \( B^P = \text{sec}^P(B) \), where the mapping \( \text{sec}^P \) transforms the LTS \( B \), leaving authorized transitions (transitions with operations and objects that are authorized by \( P \)) unmodified, and changing non-allowed transitions to transitions into an *exception state*.

**Definition 9.** Access control enforcement is a map \( \text{sec}^P : \text{LTS} \rightarrow \text{LTS} \) that, given an application design without access control \( \text{APP} = \langle \text{PI}, \text{Obj}, B \rangle \) with an LTS \( B = \langle Q, \text{PI}, \text{Obj}, T \rangle \), an active subject \( \text{as} \in \text{Subj} \), and a policy \( P \subseteq \text{Subj} \times \text{SecSubs} \times \text{Obj} \), produces a new LTS \( \text{sec}^P(B) = \langle Q, \text{PI}, \text{Obj}, T^P \rangle \) with \( T^P \) defined as follows

\[
T^P = A^P \cup D^P
\]

where

\[
A^P = \left\{ \langle q_a, \text{op}, \text{obj}, q_b \rangle \mid \langle q_a, \text{op}, \text{obj}, q_b \rangle \in T \land (\text{op} \in \text{SecSubs} \Rightarrow \langle \text{as}, \text{op}, \text{obj} \rangle \in P) \right\}
\]

and

\[
D^P = \left\{ \langle q_a, \text{op}, \text{obj}, e \rangle \mid \exists \langle q_a, \text{op}, \text{obj}, q_b \rangle \in T \land (q_a, \text{op}, \text{obj}, q_b) \notin A^P \right\}
\]

\( T^P \) includes two sets of transitions: \( A^P \) of authorized transitions and \( D^P \) of denied transitions. \( A^P \) includes: the transitions \( \langle q_a, \text{op}, \text{obj}, q_b \rangle \in T \) in which \( \text{op} \) belongs to the secure subsystem and \( P \) authorizes \( \text{op} \) and \( \text{obj} \) to \( \text{as} \), and the transitions \( \langle q_a, \text{op}, \text{obj}, q_b \rangle \in T \) in which \( \text{op} \) does not belong to the secure subsystem. \( D^P \) includes all of the transitions \( \langle q_a, \text{op}, \text{obj}, e \rangle \) corresponding to transitions forbidden to \( \text{as} \), i.e., \( \langle q_a, \text{op}, \text{obj}, q_b \rangle \notin A^P \). The state \( e \in Q \) is the exception state.
4.4 Programming Language

The design of an application is an abstract view of the application structure and behavior. When realizing a design in code, there are many more details that need to be described: control flow statements, assignments, exceptions, to name a few. This section describes a formal programming language that will be utilized by the proof of correctness to represent all of the above details in the access control code.

The main assumption is that the programming language that implements the application is imperative. The reason is that variable assignment, among other related features, are widely used by programmers. In fact, many widespread languages are imperative [1,15,18]. Moreover, the behavior of an imperative program is consistent to the application design as described in Sections 4.1 and 4.2. The basic element of an imperative language is the store, an abstraction of the memory of a computer, which keeps the values of all of the variables during the execution of a program. Variable assignments modify the values in the store. The concept of a store that is modified by executing expressions in the language is similar to the application design, as defined in Section 4.2, where the objects of the application are modified by the execution of operations. As Section 4.5 demonstrates, this similarity facilitates the comparison between design and code.

The language used herein is an extension to λ-calculus [29], which will be formalized in two steps: first, its syntax, then its semantics. The syntax will be given in standard backus-naur form (BNF). Language semantics can be specified in one of three approaches: denotational semantics, axiomatic semantics, and structural operational semantics (SOS). The flexibility and relative proximity of SOS to LTS make SOS a compelling choice.

Structural operational semantics (SOS) [30] defines the semantics in terms of a state transition system, in which states correspond to expressions in the programming language and transitions correspond to steps in the program execution that reduce those expressions. The single-step evaluation relation specifies the reduction of these expressions and is typically captured by a set of rules of the form

\[
P \quad \frac{}{\langle t, \sigma \rangle \rightarrow \langle t', \sigma' \rangle}
\]

that mean that the computation state \( \langle t, \sigma \rangle \) can be reduced to the new state \( \langle t', \sigma' \rangle \), provided that the premise \( P \) is satisfied. The computation state is captured as a pair of a program term \( t \) and a memory store \( \sigma \). The transitive closure of the single-step evaluation relation is often written \( \langle t, \sigma \rangle \xrightarrow{*} \langle t', \sigma' \rangle \), which means that \( \langle t, \sigma \rangle \) can reduce to \( \langle t', \sigma' \rangle \) in zero or more steps of the single-step evaluation relation.
→. The computation ends when \( \langle t', \sigma' \rangle \) is irreducible. When \( t' \) is irreducible, it is called a \textit{value}. For example, in a language that evaluates arithmetic expressions, the values are numbers.

Formally, a language is as follows:

**Definition 10.** A programming language is a tuple \( \langle \tau, \Sigma, \rightarrow, V \rangle \), where \( \tau \) is the set of all terms that belong to the language generated by the grammar; \( \Sigma \) is the set of all possible stores; \( \rightarrow \subseteq \tau \times \Sigma \times \tau \times \Sigma \) is a single-step evaluation relation; and, \( V \subseteq \tau \) is a set irreducible values.

If a tuple \( \langle t, \sigma \rangle \) is irreducible, and \( t \notin V \), one says that the evaluation is \textit{stuck}. A stuck term indicates that the computation has ended as an error [29]. \( \langle t, \sigma \rangle \) is \textit{well-formed} when its evaluation does not get stuck, i.e., \( \langle t, \sigma \rangle \xrightarrow{\ast} \langle v, \sigma' \rangle \) and \( v \in V \).

The \( \lambda \)-calculus described in this section is untyped, which means that there are no rules to verify that the terms in the language are well-formed. Indeed, proving security assurance does not require a typed language and the implication greatly eases the presentation. To provide security assurance, one just need to prove that the body of a method executes only if the policy authorizes that method to a subject.

In the following, let \( \text{Lang} = \langle \tau, \Sigma, \rightarrow, V \rangle \) represent the extended \( \lambda \)-calculus language. Figure 11 details the syntax of \( \text{Lang} \). The notation is as follows: \( t \) is a non-terminal, which denotes all of the valid expressions in the grammar of the language; the non-terminal \( v \) represents values that belong to \( V \), i.e., irreducible expressions; and, \( x \) and \( y \) are terminals that correspond to identifiers.

\( \text{Lang} \) features several constructs: methods, branching, variable assignments and dereference, classes, instances, and exceptions. The production \( t ::= \lambda x.t \) corresponds to a \( \lambda \)-function definition, where \( x \) is the identifier of the input parameter of the function and \( t \) is the function body. The production \( t ::= tt \) corresponds to an \textit{application}, i.e., a function (method) call, where the first \( t \) is the callee and the second \( t \) is the argument.

\( \text{Lang} \) can also manipulate and access a store \( \sigma \). A store is an abstraction of the memory of a computer. In a real-world program, the places in the memory are referenced by numbers. Similarly, a store \( \sigma \) is a mapping between memory locations and values. For example, the expression \( \sigma(l) \) is the value referenced by location \( l \) in store \( \sigma \). The production \( t ::= t := t \) corresponds to \textit{assignment}, where the first \( t \) corresponds to a location and the second \( t \) corresponds to the value to store at that location. The production \( t ::=!t \) corresponds to \textit{dereference}, which retrieves the value referenced by the location \( t \) in the store.

The production \( t ::= \text{exception} \) corresponds to exceptions. Any term containing an exception evaluates into an exception. For example, \( (\lambda x.x \text{ true}) \text{ exception} \) evaluates into \text{exception}. 

16
The single step evaluation relation for $\text{Lang}$ is defined as a set of rules shown in Figures 12 and 13. Rules with an empty premise are denoted as $Q$, i.e., without the horizontal line. Rules (1), (2), and (3) evaluate applications. The notation $[x/v_2]t_{12}$ of rule (3) means “the term resulting from replacing all occurrences of $x$ in $t_{12}$ by $v_2$,” and corresponds to beta-reduction in $\lambda$-calculus. Rules (4) and (5) evaluate reference creation. The notation $\sigma \cup \{l \mapsto v_1\}$ in rule (4) means “a store that contains the same locations and values as $\sigma$, and also maps $l$ into $v_1$.” Rules (6) and (7) evaluate dereferencing. The expression $\sigma(l)$ in (6) is the value referenced by $l$ in store $\sigma$. Rules (8) and (9) evaluate variable assignments. The expression $\sigma[l \mapsto v_2]$ in rule (8) means “the store that maps $l$ into $v_2$ and all of the other locations map the same as in $\sigma$.” Rules (10), (11), and (12) evaluate if statements.

Figure 13 details the evaluation rules for exceptions. In a real-world language, when an exception occurs, there are memory locations that change to indicate that the program reached an exception state. For simplicity assume that, whenever there is an exception, the store changes into an exception store $\sigma_e$, whose internal structure is not detailed.

4.5 Application Code

The language of Section 4.4, $\text{Lang}$, implements the application at the code level. To facilitate the comparison between the code and the design, this section details an approach to transform the application code into an LTS that can be directly compared with the application design.
\[
\begin{align*}
\langle t_1, \sigma \rangle &\rightarrow \langle t_1', \sigma' \rangle \\
\langle t_1 \ t_2, \sigma \rangle &\rightarrow \langle t_1' \ t_2, \sigma' \rangle \\
\langle t_2, \sigma \rangle &\rightarrow \langle t_2', \sigma' \rangle \\
\langle v_1 \ t_2, \sigma \rangle &\rightarrow \langle v_1' \ t_2, \sigma' \rangle \\
\langle \lambda x. t_{12}, \sigma \rangle &\rightarrow \langle x/v_2 \ t_{12}, \sigma \rangle \\
\langle t_2, \sigma \rangle &\rightarrow \langle t_2', \sigma' \rangle \\
\langle v_1 \ t_2, \sigma \rangle &\rightarrow \langle v_1' \ t_2, \sigma' \rangle \\
\langle \text{ref } v_1, \sigma \rangle &\rightarrow \langle \text{ref } v_1', \sigma' \rangle \\
\langle \text{ref } t_1, \sigma \rangle &\rightarrow \langle \text{ref } t_1', \sigma' \rangle \\
\langle l := v_2, \sigma \rangle &\rightarrow \langle \text{unit}, \sigma[v \mapsto v_2] \rangle \\
\langle l \notin \text{dom}(\sigma) \rangle &\rightarrow \langle 1, \sigma \cup [1 \mapsto v_1] \rangle \\
\langle t_1, \sigma \rangle &\rightarrow \langle t_1', \sigma' \rangle \\
\langle \text{ref } t_1, \sigma \rangle &\rightarrow \langle \text{ref } t_1', \sigma' \rangle \\
\sigma(l) = v &\rightarrow \langle l, \sigma \rangle \rightarrow \langle v, \sigma \rangle \\
\langle l := t_2, \sigma \rangle &\rightarrow \langle l' := t_2, \sigma' \rangle \\
\langle \text{if true then } t_2 \text{ else } t_3, \sigma \rangle &\rightarrow \langle t_2, \sigma \rangle \\
\langle \text{if false then } t_2 \text{ else } t_3, \sigma \rangle &\rightarrow \langle t_3, \sigma \rangle \\
\langle \text{exception } t_2, \sigma \rangle &\rightarrow \langle \text{exception}, \sigma_e \rangle \\
\langle v_1 \text{ exception}, \sigma \rangle &\rightarrow \langle \text{exception}, \sigma_e \rangle \\
\langle \text{ref exception}, \sigma \rangle &\rightarrow \langle \text{exception}, \sigma_e \rangle \\
\langle \text{exception} := t_2, \sigma \rangle &\rightarrow \langle \text{exception}, \sigma_e \rangle \\
\langle v_1 := \text{exception}, \sigma \rangle &\rightarrow \langle \text{exception}, \sigma_e \rangle
\end{align*}
\]
Recall that at the design level, the application comprises a set of objects and a set of operations (public interface) that access those objects. The behavior of the application is an LTS, where the states are sets of objects of the application at a given point in time, and transitions are triggered by the invocation of an operation over an object. The standard way to realize such structure in code is to use a programming language to implement each operation as a function, method, or procedure, and each object as an instance of a class or basic data type in the language [16,19,21]. In the context of Lang this means that there is a set of λ-functions to represent the operations and a set of values (irreducible terms of Lang) represent the objects. Formally, this is represented as two mappings: an operation mapping, which maps operations in the public interface of the design into terms of Lang; and, an object mapping, which maps objects of the design into values of Lang.

Definition 11. An operation mapping is a mapping $\text{IOp} : \text{PI} \rightarrow \tau$ between operations in the public interface and λ-functions in Lang that realize them.

Definition 12. An object mapping is a mapping $\text{IObj} : \text{Obj} \rightarrow \mathbb{V}$ between objects in the application design and values of Lang.

To simplify, $t_{\text{op}}$ represents the term mapped from $\text{op}$, i.e., $t_{\text{op}} = \text{IOp}(\text{op})$, and $v_{\text{obj}}$ represents the value mapped from $\text{obj}$, i.e., $v_{\text{obj}} = \text{IObj}(\text{obj})$.

To transform the code specified by $\text{IOp}$ and $\text{IObj}$ into an LTS, this section takes advantage of the similarity between the formalisms for the application design and code. Programs in Lang, as the LTS of the design, are state transition systems, where states correspond to tuples of the form $\langle t, \sigma \rangle$ ($t$ is a term and $\sigma$ is a store), and transitions correspond to evaluations of the form $\langle t, \sigma \rangle \rightarrow \langle t', \sigma' \rangle$. The difference between a program in Lang and an LTS of the design is that the latter does not detail what happens during the execution of an operation. For each operation execution, the LTS of the design details only the transition between the states that are right before and after the execution of each operation, and has no additional states detailing what happens in between. Therefore, to transform the code into an LTS, one must abstract the evaluation of the body of each λ-function, meaning that the resulting LTS does not contain intermediate states associated to terms yielded during the evaluation of a λ-function. This section defines the implementation of an application $\text{APP}$ (see Section 8) in a language Lang as an entity $\text{IAPP}_\text{Lang}$ that: uses $\text{IOp}$ to map each operation from the design into λ-functions; uses $\text{IObj}$ to map objects into values of Lang; and, has a behavior, represented as an LTS, where the states correspond to stores $\sigma$ of Lang and the transitions correspond to the evaluation of λ-functions. In particular, the
transition from a state $\sigma$ to $\sigma'$ triggered by the invocation of an operation $op$ over an object $obj$, belongs to the transitions of the LTS if and only if the execution of the term associated to $op$ (i.e., $t_{op} = IOp(op)$) on the concrete receiver associated to $obj$ (i.e., $v_{obj} = IObj(obj)$) in the context of the store $\sigma$, evaluates through the evaluation relation $\xrightarrow{*}$ to an irreducible value $v' \in V$, and the store $\sigma'$.

**Definition 13.** An implementation $I_{APP,Lang}$ of an application design $APP$ in a language $Lang$ is a tuple $\langle IOp, IObj, IB \rangle$, where

- $IOp : PI \rightarrow \tau$ is an operation mapping.
- $IObj : Obj \rightarrow V$ is an object mapping.

$IB$ is an LTS of the form $\langle IQ, PI, Obj, IT \rangle$, where $IQ = \Sigma$ are the states of the system and $IT \subseteq IQ \times PI \times Obj \times IQ$ is the transition relation, where

$$IT = \{ \langle \sigma, op, obj, \sigma' \rangle | \langle t_{op}(v_{obj}), \sigma \rangle \xrightarrow{*} \langle v', \sigma' \rangle, v' \in V \}$$

### 4.6 Correct Implementation of an Application Design

To prove that an implementation is correct with respect to an application design, one must demonstrate that the LTS of the application code is consistent with the LTS of the application design. The use of simulation relations [23] is an approach to compare labeled transition systems and determine if their behavior is consistent. Given two LTS $\langle QA, PI, Obj, TA \rangle$ and $\langle QB, PI, Obj, TB \rangle$, a simulation relation is a relation $S \subseteq QA \times QB$ that satisfies the following condition: for every pair of states $\langle qa, qb \rangle$ that belong to the simulation relation, if there is a transition $\langle qa, op, obj, qa' \rangle$ in $TA$, then there exist a transition $\langle qb, op, obj, qb' \rangle$ in $TB$ where $\langle qa', qb' \rangle$ also belongs to the simulation relation.

**Definition 14.** A simulation relation between two labeled transition systems $\langle QA, PI, Obj, TA \rangle$, $\langle QB, PI, Obj, TB \rangle$ is a relation $S \subseteq QA \times QB$ such that, for all $\langle qa, qb \rangle \in S$, if $\langle qa, op, obj, qa' \rangle \in TA$, then there exists $qb' \in QB$ with $\langle qb, op, obj, qb' \rangle \in TB$ and $\langle qa', qb' \rangle \in S$.

If applied for the LTS of the design and code, a simulation relation yields a many-to-many correspondence between states in both transition systems. The implication is that, for each pair of states in the LTS of the design connected by a transition, there exist one or more pairs of states in the LTS of the code connected by a corresponding transition, and vice versa. From an access control perspective, this is an undesirable situation, since a proof of correctness must ensure that: no method execution at the code
A correct implementation of the application design means that there exist a strong simulation relation between the LTS of the code and the LTS of the design.

Definition 16. An implementation $I_{\text{APP, Lang}} = \langle I\text{Op}, I\text{Obj}, I\text{B} \rangle$ correctly implements an application $\text{APP} = \langle P\text{I}, \text{Obj}, B \rangle$ if and only if there exist a strong simulation relation between $B$ and $I\text{B}$.

4.7 Scope of the Proof

The program that realizes access control at the code level can be very complex. For instance, the code of Section 3 relies on libraries, each one with specific semantics that must be considered for the proof of correctness. When providing security assurance, it is crucial to define the scope of the proof, i.e., which parts of the code will be directly included in the proof, and which parts will be abstracted. The parts of the code within the scope of the proof must be included as statements in the formal programming
language of choice (e.g., \textit{Lang}). The remainder of the code must be abstracted as additional rules of evaluation in the semantics of the language.

The decision of the scope for the proof of correctness depends on the particular requirements of each application. However, the process to provide assurance is the same, regardless of the scope. To better describe the proof, this section focuses only on security assurance for the enforcement code itself. Therefore, the proof of correctness abstracts the behavior of the code that checks for permissions into Rules (20) and (21), which define the semantics of the operation \texttt{isAuthorized} to check if the active subject can execute a term \( t \) over a value \( v \). Recall that a policy \( P \subseteq \text{Subj} \times \text{SecSubs} \times \text{Obj} \) (see Section 4.1) is a set of authorizations for subjects to invoke operations of the secure subsystem \( \text{SecSubs} \subseteq \text{PI} \) over objects in the application. The assumption for these rules is that the stores represent the active subject as \( \sigma(l_{\text{as}}) \), so the expression \( \langle \text{IObj}^{-1}(\sigma(l_{\text{as}})), \text{IOP}^{-1}(t), \text{IObj}^{-1}(v) \rangle \in P \) means that the subject that maps to \( \sigma(l_{\text{as}}) \) is authorized to invoke the operation that maps to term \( t \), over the object that maps to value \( v \).

\[
\begin{align*}
\frac{\langle \text{IObj}^{-1}(\sigma(l_{\text{as}})), \text{IOP}^{-1}(t), \text{IObj}^{-1}(v) \rangle \in P}{\langle \text{isAuthorized } tv, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle} & \quad (20) \\
\frac{\langle \text{IObj}^{-1}(\sigma(l_{\text{as}})), \text{IOP}^{-1}(t), \text{IObj}^{-1}(v) \rangle \notin P}{\langle \text{isAuthorized } tv, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle} & \quad (21)
\end{align*}
\]

5 Providing Security Assurance to Enforcement Code

To validate the proposed approach for security assurance, this section utilizes the access control enforcement approaches that are part of the university application case study (see Section 3). This section formalizes each enforcement approach utilizing imperative \( \lambda \)-calculus and then proves the existence of a strong simulation relation between the LTS yielded by each enforcement code and the LTS of the behavior of the application at the design level.

Section 5.1 proves correctness of the basic enforcement. Section 5.2 proves correctness of the aspect-oriented enforcement.

5.1 Assurance for the Basic Enforcement

As described in Section 3.1, the basic enforcement consists of manually modifying methods in the secure subsystem (the methods that need protection) to incorporate enforcement code. For instance, if the callee that needs protection is a term \( t \), then the basic enforcement changes \( t \) to
\lambda y. \text{if } \text{isAuthorized } t \ y \ \text{then } (t \ y) \ \text{else exception} \\

The new callee (22) wraps the original callee \( t \) into an \texttt{if} statement that uses the \texttt{isAuthorized} operation (defined by Rules (20) and (21)) to check whether \( t \) can be called or not, and execute \( t \) only if allowed by the policy.

The proof of correctness for the basic enforcement code has several elements. There is an application design \( \text{APP} = \langle \text{PI, Obj, B} \rangle \) that realizes the main concern of the application without access control and an implementation \( \text{I}_{\text{APP, Lang}} = \langle \text{IOp, IObj, IB} \rangle \) that correctly implements \( \text{APP} \). This means that there exist a strong simulation relation between \( B \) and \( IB \) (see Definition 16). \( \text{APP}^P = \langle \text{PI, Obj, sec}^P(B) \rangle \) is the result of modifying the behavior of \( \text{APP} \) to enforce a policy \( P \). At the code level, \( \text{Basic}(\text{I}_{\text{APP, Lang}}) \) is the result of modifying every method in the secure subsystem to incorporate enforcement code. To provide security assurance, one must prove that there exist a strong simulation relation between the behaviors of \( \text{APP}^P \) and \( \text{Basic}(\text{I}_{\text{APP, Lang}}) \).

Formally, the basic enforcement is a mapping \( \text{Basic} \) that transforms an implementation \( \text{I}_{\text{APP, Lang}} = \langle \text{IOp, IObj, IB} \rangle \) into a new implementation \( \text{Basic}(\text{I}_{\text{APP, Lang}}) \) with access control. To modify the callees that need access control, the basic enforcement defines a new operation mapping \( \text{IOp}^{\text{Basic}} \) that adds permission checking as shown in (22) to the callees mapped by \( \text{IOp} \). The basic enforcement defines a new LTS, where the states are the same as in \( IB \), and the transitions are obtained from the evaluation of the terms obtained from \( \text{IOp}^{\text{Basic}} \). Specifically, the transition from a state \( \sigma \) to \( \sigma' \) triggered by the invocation of an operation \( op \) over an object \( obj \), belongs to the transitions of the new LTS if and only if the execution of the term associated to \( op \) (i.e., \( t_{op}^{\text{Basic}} = \text{IOp}^{\text{Basic}}(op) \)) on the concrete receiver associated to \( obj \) (i.e., \( v_{obj} = \text{IObj}(obj) \)) in the context of the store \( \sigma \), evaluates through the evaluation relation \( \xrightarrow{*} \) to an irreducible value \( v' \in V \), and the store \( \sigma' \).

**Definition 17.** Basic enforcement is a mapping \( \text{Basic}(\text{I}_{\text{APP, Lang}}) = \langle \text{IOp}^{\text{Basic}}, \text{IObj, IB}^{\text{Basic}} \rangle \) which, given an implementation \( \text{I}_{\text{APP, Lang}} = \langle \text{IOp, IObj, IB} \rangle \) with a behavior \( \text{IB} = \langle \text{IQ, PI, Obj, IT} \rangle \), satisfies the following:

\[
\text{IOp}^{\text{Basic}}(op) = \begin{cases} 
\lambda y. \text{if } \text{isAuthorized } t_{op} \ y \ \text{then } (t_{op} \ y) \ \text{else exception} & \text{; if } \text{op} \in \text{SecSubs} \\
 t_{op} & \text{; if } \text{op} \notin \text{SecSubs} 
\end{cases}
\]

\( \text{IB}^{\text{Basic}} = \langle \text{IQ, PI, Obj, IT}^{\text{Basic}} \rangle \)
\[ IT^{Basic} = \left\{ \langle \sigma, \text{op}, \text{obj}', \sigma' \rangle | \langle t_0^{Basic} \cdot \text{v}_\text{obj}, \sigma \rangle \xrightarrow{\ast} \langle \text{v}', \sigma' \rangle, \text{v}' \in V \right\} \]

The following lemma proves that the callees instrumented with enforcement code evaluate into the original callees if and only if authorized by the policy, or evaluate into an exception otherwise.

**Lemma 18.** Given a policy \( P \); an active subject \( as \); and, an implementation \( I_{\text{APP}, \text{Lang}} \), where all of the stores of \( B \) represent the active subject as \( \sigma(l_{as}) \). The following statements are true:

\[
\begin{align*}
&\quad\langle \text{if} (\text{isAuthorized} \cdot \text{op} \cdot \text{v}) \text{ then } t_1 \text{ else } t_2, \sigma \rangle \xrightarrow{\ast} \langle t_1, \sigma' \rangle \\
&\iff \langle \text{IObj}^{-1}(\sigma(l_{as})), \text{IOp}^{-1}(\text{op}), \text{IObj}^{-1}(\text{v}) \rangle \in P
\end{align*}
\]

\[
\begin{align*}
&\quad\langle \text{if} (\text{isAuthorized} \cdot \text{op} \cdot \text{v}) \text{ then } t_1 \text{ else } t_2, \sigma \rangle \xrightarrow{\ast} \langle t_2, \sigma' \rangle \\
&\iff \langle \text{IObj}^{-1}(\sigma(l_{as})), \text{IOp}^{-1}(\text{op}), \text{IObj}^{-1}(\text{v}) \rangle \notin P
\end{align*}
\]

**Proof.** According to Rule (20), \( \langle \text{isAuthorized} \cdot \text{op} \cdot \text{v}, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle \) if \( \langle \text{IObj}^{-1}(\sigma(l_{as})), \text{IOp}^{-1}(\text{op}), \text{IObj}^{-1}(\text{v}) \rangle \in P \). Using rule (10), the if statement evaluates into \( t_1 \) when the condition is true.

According to Rule (21), \( \langle \text{isAuthorized} \cdot \text{op} \cdot \text{v}, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle \) if \( \langle \text{IObj}^{-1}(\sigma(l_{as})), \text{IOp}^{-1}(\text{op}), \text{IObj}^{-1}(\text{v}) \rangle \notin P \). Using rule (11), the if statement evaluates into \( t_2 \) when the condition is false. \( \square \)

The theorem of correctness for the basic access control enforcement is as follows:

**Theorem 19.** Given a policy \( P \) with an active subject \( as \), an application design without access control \( \text{APP} = \langle \text{PI}, \text{Obj}, B \rangle \) with behavior \( B = \langle Q, \text{PI}, \text{Obj}, T \rangle \), an application design with access control \( \text{APP}^P = \langle \text{PI}, \text{Obj}, \text{sec}^P(B) \rangle \) with behavior \( \text{sec}^P(B) = \langle Q, \text{PI}, \text{Obj}, T^P \rangle \), an implementation without access control \( I_{\text{APP}, \text{Lang}} = \langle \text{IOp}, \text{IObj}, \text{IB} \rangle \) with behavior \( \text{IB} = \langle IQ, \text{PI}, \text{Obj}, IT \rangle \), an implementation with access control \( \text{Basic}(I_{\text{APP}, \text{Lang}}) = \langle \text{IOp}^{Basic}, \text{IObj}, \text{IB}^{Basic} \rangle \) with behavior \( \text{IB}^{Basic} = \langle IQ, \text{PI}, \text{Obj}, IT^{Basic} \rangle \), and a strong simulation relation \( S \) between \( B \), and \( \text{IB} \). \( S \) is also a strong simulation relation between \( \text{sec}^P(B) \) and \( \text{IB}^{Basic} \).

**Proof.** From the definition of strong simulation relation (Definition 15), for all \( q, \text{op}, \text{obj} \) where \( T \) is defined, it is true that

\[
S(T(q, \text{op}, \text{obj})) = IT(S(q, \text{op}, \text{obj}))
\]

(25)
What must be proven is that, for all $q, op, obj$ where $T^P$ is defined

$$S(T^P(q, op, obj)) = IT^{Basic}(S(q), op, obj)$$  \hspace{1cm} (26)$$

There are three different cases. To simplify notation: $S(q) = \sigma_q$ is the store mapped from a state $q$ by $S$ and $S(e) = \sigma_e$ is the exception store mapped from the exception state by $S$.

**Case 1.** $op \notin SecSubs$

From Definition 9, all of the transitions in $T^P$ where $op \notin SecSubs$ are authorized, thus

$$T^P(q, op, obj) = T(q, op, obj)$$  \hspace{1cm} (27)$$

From Definition 17 and since $op \notin SecSubs$, it follows that

$$IOp(op) = IOp^{Basic}(op)$$  \hspace{1cm} (28)$$

and the transitions associated to those callees do no change (see Definition 13), thus

$$IT = IT^{Basic}$$  \hspace{1cm} (29)$$

Since $T^P = T$, $IT = IT^{Basic}$, and

$$S(T(q, op, obj)) = IT(S(q), op, obj)$$

it follows that

$$S(T^P(q, op, obj)) = IT^{Basic}(S(q), op, obj)$$

is true for this case.

**Case 2.** $op \in SecSubs \land \langle as, op, obj \rangle \in P$

From Definition 9, all of the transitions in $T^P$ where $op \in SecSubs \land \langle as, op, obj \rangle \in P$ are authorized, thus

$$T^P(q, op, obj) = T(q, op, obj)$$  \hspace{1cm} (30)$$
From Definition 17 and since \( op \in \text{SecSubs} \), it follows that 

\[
I_{\text{op}}^{\text{Basic}}(op) = \lambda y. \text{if } \text{isAuthorized } t_{\text{op}} y \text{ then } (t_{\text{op}} y) \text{ else exception}
\]  

(31)

By Lemma 18 and since \( \langle \text{as}, \text{op}, \text{obj} \rangle \in P \), it follows that 

\[
\left\langle \begin{array}{l}
\text{if } \text{isAuthorized } t_{\text{op}} v_{\text{obj}} \\
\text{then } (t_{\text{op}} v_{\text{obj}}) \text{ else exception}
\end{array} \right\rangle^* \rightarrow \langle t_{\text{op}} v_{\text{obj}}, \sigma'_{q} \rangle
\]  

(32)

and the corresponding transitions in the LTS do not change (see Definition 17), thus

\[ IT = I_{T}^{\text{Basic}} \]  

(33)

Since \( T_{P} = T \), \( IT = I_{T}^{\text{Basic}} \), and

\[ S(T(q, op, obj)) = IT(S(q), op, obj) \]

it follows that

\[ S(T_{P}(q, op, obj)) = IT^{\text{Basic}}(S(q), op, obj) \]

is true for this case.

Case 3. \( op \in \text{SecSubs} \land \langle \text{as}, \text{op}, \text{obj} \rangle \notin P \)

From Definition 9, all of the transitions in \( T_{P} \) where \( op \in \text{SecSubs} \) and \( \langle \text{as}, \text{op}, \text{obj} \rangle \notin P \) are denied, thus

\[ T_{P}(q, op, obj) = e \]  

(34)

where \( e \) is the exception state.

From Definition 17 and since \( op \in \text{SecSubs} \), it follows that 

\[
I_{\text{op}}^{\text{Basic}}(op) = \lambda y. \text{if } \text{isAuthorized } t_{\text{op}} y \text{ then } (t_{\text{op}} y) \text{ else exception}
\]  

(35)

From Lemma 18 and since \( \langle \text{as}, \text{op}, \text{obj} \rangle \notin P \), it follows that
\[
\begin{cases}
\text{if} \ (\text{isAuthorized to } v_{\text{obj}}) \\
\text{then} \ (t_{\text{op}} v_{\text{obj}}) \text{ else exception}
\end{cases}, \sigma_q \mapsto \langle \text{exception}, \sigma_e \rangle
\] (36)

From Definition 17 it follows that

\[ IT_{\text{Basic}}(\sigma_q, \text{op}, \text{obj}) = \sigma_e = S(e) \] (37)

Since \( T^P(q, \text{op}, \text{obj}) = e \), \( IT_{\text{Basic}}(\sigma_q, \text{op}, \text{obj}) = S(e) \), and

\[ S(T(q, \text{op}, \text{obj})) = IT(S(q), \text{op}, \text{obj}) \]

it follows that

\[ S(T^P(q, \text{op}, \text{obj})) = IT_{\text{Basic}}(S(q), \text{op}, \text{obj}) \]

is true for this case.

\[ \square \]

### 5.2 Assurance for the Aspect-Oriented Enforcement

To provide security assurance for the aspect-oriented enforcement, one must first formalize aspects and the weaving process. The access control aspect of Section 3.2 contains only one pointcut definition, and one advice. For simplicity, this section formalizes aspects similarly. To formalize the execution join points utilized by the access control aspect, the weaver directly instruments the terms in \( \text{Lang} \) that implement the operations from the design, not the function invocations within those terms. The special sentence \text{proceed} [4] is formalized through a \( \lambda \)-function \( \text{adv} = \lambda x.t \), where \( x \) represents the \text{proceed} sentence inside \( t \).

**Definition 20.** Given an application design \( \text{APP} = (\text{PI}, \text{Obj}) \), and an implementation \( I_{\text{APP,Lang}} = (\text{IOp, IObj, IB}) \), an aspect is a tuple \( (\text{PC}, \text{adv}) \), where \( \text{PC} \subseteq \text{PI} \) is the set of operations to be modified by the aspect weaver (i.e., pointcut definition); and, \( \text{adv} = \lambda x.t \) is a \( \lambda \)-function that transforms the terms that \( \text{IOp} \) maps from the operations in \( \text{PC} \) (i.e., the advice).

Aspect weaving is a mapping that takes as arguments: an aspect \( (\text{PC}, \text{adv}) \) and an implementation
\langle \text{IOp}, \text{IObj}, \text{IB} \rangle$, and returns a new implementation \langle \text{IOp}^W, \text{IObj}, \text{IB}^W \rangle. For each operation that is not instrumented by the aspect (i.e., not referenced by the pointcut definition of the aspect), \text{IOp}^W maps to the same term as \text{IOp}. For the operations referenced by the pointcut definition, \text{IOp}^W maps to the application of \text{adv} into the original term \text{top} mapped by \text{IOp}. \text{IB}^W is a new LTS that represents the behavior obtained from the terms mapped by \text{IOp}^W, as specified from the definition of an implementation (Definition 13).

**Definition 21.** Weaving is a mapping \text{W} that takes as arguments an aspect, and an implementation, and it yields a new implementation with the aspect woven into it.

\[
\text{W}(\langle \text{PC}, \text{adv} \rangle, \langle \text{IOp}, \text{IObj}, \text{IB} \rangle) = \langle \text{IOp}^W, \text{IObj}, \text{IB}^W \rangle
\]

where

\[
\text{IOp}^W(\text{op}) = \begin{cases} \text{adv} \text{top} & \text{if } \text{op} \in \text{PC} \\ \text{top} & \text{if } \text{op} \notin \text{PC} \end{cases}
\]

\[
\text{IB}^W = \langle \text{IQ}, \text{PI}, \text{Obj}, \text{IT}^W \rangle
\]

\[
\text{IT}^W = \left\{ \langle \sigma, \text{op}, \text{obj}, \sigma' \rangle | \langle \text{top}^W \text{v}_{\text{obj}}, \sigma \rangle \xrightarrow{\sigma} \langle \text{v}', \sigma' \rangle, \text{v}' \in \text{V} \right\}
\]

Using the above definitions, the access control aspect of Section 3.2 is as follows:

\[
\text{A}_{\text{AC}} = \left\langle \text{SecSubs}, \lambda \text{y}. \text{if (isAuthorized } \text{x,y) then (x,y) else exception} \right\rangle
\]

The access control aspect (38) has a pointcut definition that references all of the methods in the secure subsystem, and has an advice that wraps the original callees with permission checking code. This is very similar to the basic enforcement, since both approaches add a layer of code to protect the methods in the secure subsystem. This section introduces a lemma that proves that the implementation yielded by the aspect-oriented enforcement is equivalent to the implementation yielded by the basic enforcement. As a result, the same proof of correctness can again be fully reused.

**Lemma 22.** Given the access control aspect \(A_{\text{AC}}\),

\[
\text{Basic}(I_{\text{APP,Lang}}) = \text{W}(A_{\text{AC}}, I_{\text{APP,Lang}}) \text{ for all implementations } I_{\text{APP,Lang}} = \langle \text{IOp}, \text{IObj}, \text{IB} \rangle.
\]
Proof. From the definition of weaving (Definition 21), the resulting implementation has an operation mapping $IOp^W$ that maps to different terms, depending on whether the operation is referenced by the pointcut definition or not. Therefore, there are two cases:

Case 1. $op \in PC$

From the definition of weaving (Definition 21), it follows that

$$IOp^W(op) = [x/t_{op}]adv$$

Since the access control aspect is

$$A_{AC} = \left\langle \text{SecSubs}, \lambda y. \text{if} (\text{isAuthorized } x, y) \THEN (x, y) \text{ else exception} \right\rangle$$

it follows that

$$IOp^W(op) = \lambda y. \text{if} (\text{isAuthorized } t_{op}, y) \THEN (t_{op}, y) \text{ else exception}$$

and $PC = \text{SecSubs}$.

From the definition of basic enforcement (Definition 17) and since $op \in \text{SecSubs}$, it follows that

$$IOp^{Basic}(op) = \lambda y. \text{if} (\text{isAuthorized } t_{op}, y) \THEN (t_{op}, y) \text{ else exception}$$

which proves this case.

Case 2. $op \notin PC$

From the definition of weaving (Definition 21), it follows that

$$IOp^W(op) = t_{op}$$
Since the access control aspect is

\[ A_{AC} = \langle \text{SecSubs}, \lambda y. \text{if (isAuthorized x y) then (x y) else exception} \rangle \]

it follows that \( PC = \text{SecSubs} \).

From the definition of basic enforcement (Definition 17) and since \( op \in \text{SecSubs} \), it follows that

\[ \text{IOp}^{\text{Basic}}(op) = t_{op} \]

which proves this case.

The theorem of correctness for the aspect-oriented access control enforcement is as follows:

**Theorem 23.** Given a policy \( P \) with an active subject \( as \); an application design without access control \( \text{APP} = \langle PI, Obj, B \rangle \) with behavior \( \langle Q, PI, Obj, T \rangle \); an application with access control \( \text{APP}^P = \langle PI, Obj, IB \rangle \) with behavior \( \text{sec}^P(B) = \langle Q, PI, Obj, T^P \rangle \); an implementation without access control \( \text{I}_{\text{APP,Lang}} = \langle \text{IOp}, \text{IObj}, \text{IB} \rangle \) with behavior \( IB = \langle IQ, PI, Obj, IT \rangle \); an implementation with access control \( W(A_{AC}, I_{\text{APP,Lang}}) = \langle \text{IOp}^{AO}, \text{IObj}, \text{IB}^{AO} \rangle \) with behavior \( IB^{AO} = \langle IQ, PI, Obj, IT^{AO} \rangle \); and, a strong simulation relation \( S \) between \( B \), and \( IB \). \( S \) is also a strong simulation relation between \( \text{sec}^P(B) \) and \( IB^{AO} \).

**Proof.** From Lemma 22, the aspect-oriented enforcement \( W(A_{AC}, I_{\text{APP,Lang}}) \) is equivalent to the basic enforcement \( \text{Basic}(I_{\text{APP,Lang}}) \), for all implementations \( I_{\text{APP,Lang}} \). Therefore, Theorem 19 applies. \( \square \)

6 Related Work

There are several approaches that address access control at the code level and/or try to provide security assurance. The approaches in [12,13,27] provide mechanisms to enforce access control at the code level by modifying Java programs before or during compilation. However, they are not integrated with any model at the design level, thus they do not provide any proof of correctness for the access control enforcement code.
Other approaches utilize aspect-oriented programming for access control. Mourad et al. [24] propose a high-level aspect-oriented framework to enforce security. The main element of their approach is a language, SHL, utilized to systematically harden security into the code by generating aspect code. Alhadidi et al. [3] proposes an aspect-oriented calculus for security. Dantas [7] proposes a textual language, AspectML, to represent aspect-oriented code enforcement of access control policies. These approaches address access control at the code level, without explicit links to design models. De Win [8], Huang et al. [17] Shah et al. [32], Slowikowski et al. [33], Sewe [31], and Viega et al. [38] also utilize aspect-oriented programming to intercept methods and constrain access based on permissions. None of these approaches provide proofs of correctness with respect to design models.

At the design level, there are some approaches that address access control. The work in Basin et al. [5] proposes the SecureUML notation to specify access control as part of the main design of the application, and automatically generate access control infrastructures based on the design models. However, they only provide an informal proof of correctness for the generated code. The work of Doan et al. [10] proposes UML extensions to support RBAC and MAC, with consistency-checking rules to provide security assurance at the design. AuthUML [2] is a framework to specify access control at the use-case level. AuthUML utilizes logic programming to insure that access control is consistent, complete and without conflicts before the design and implementation stages. The work of Song et al. [34] utilizes aspect-oriented modeling to compose access-control into a class model that allows designers to verify access-control properties. Since Doan et al., AuthUML, and Song et al. approaches provide a degree of security assurance at the design level, they complement our work, which provides security assurance at the code level.

7 Conclusions and Future Work

This paper presented a framework to provide security assurance as a proof that the code that realizes access control is a correct implementation of the access control policy defined at the design level. The utilization of labeled transition systems (LTS) at the design level can represent the behavior of the application in a manner similar to a UML state diagram. At the code level, the utilization of \( \lambda \)-calculus captures the essential elements of access control that require analysis and the mappings from \( \lambda \)-calculus to LTS facilitates the comparison between design and code through a common formalism.

In addition, this paper validated the approach, applying the proof of correctness to two access control enforcement approaches, which are part of our previous work. Although this paper focuses mainly on the
access control enforcement code, this paper also demonstrates that the scope of the proofs of correctness can be extended to cover more parts of the access control code (e.g., policy code), if required.

To further improve this framework, future research considers the integration with Doan et al. work [10], which addresses access control at the requirements and design level with a focus on use cases, class, and sequence diagrams. The expected result is a comprehensive framework that integrates access control from requirements specification down to code; and, it provides security assurance both at the design and coding levels.

References


