Declarative Access Policies based on Objects, Relationships, and States

Simin Chen

University of Texas at Austin, Carnegie Mellon University

simin.chen@utexas.edu

Abstract

Access policies are hard to express in existing programming languages. However, their accurate expression is a prerequisite for many of today's applications. We propose a new language that uses classes, first-class relationships, and first-class states to express access policies in a more declarative and fine-grained way than existing solutions allow.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design – Representation; D.3.2 [*Programming Languages*]: Language Classifications – Object-oriented languages

Keywords State, Relationship, Protocol, Access policies

1. Introduction

Access control is vital to create secure mobile web applications. A dominant model for access control is Fischer et al.'s Object-sensitive Role-Based Access Control (ORBAC) [3]. ORBAC is an extension of role-based access control [5] and allows for more fine-grained access. Fine-grained access policies restrict access to shared data accurately and thus increase the security of applications. For example, when a professor sets the grade of a student in a university grading system, we may want to ensure that the professor can only set of the grade of students in the professor's class. With ORBAC, this would be achieved by assigning unique id fields to each student. The professor would be allowed to set the grade of a student if the professor has a role *ProfessorOf* with the student's id number. However, if we also wished to guarantee that grades can only be set during the grading period of a college, id fields would not be enough; we would need information about the state [6] of the grade as well.

Copyright is held by the author/owner(s).

SPLASH'12, October 19–26, 2012, Tucson, Arizona, USA.
ACM 978-1-4503-1563-0/12/10.

```
1  class Student {...}
2  class Class {...}
3  class Professor {...}
4  class College {...}
5  class Dean {...}
6  relationship Attends<Student stu, Class cl> {
7      char grade;  }
8  relationship Teaches<Professor prof, Class cl> {...}
9  relationship Employed<Professor prof, College col>
10     {...}
```

Figure 1. Classes and Relationships

In order to come up with general model of ORBAC, we first compiled a requirements document for a real-world university grading and evaluation system of a mobile web application. This document demonstrated that understanding the *entities* in the system, the *relationships* between them, and the *states* of the system is necessary to express access policies in a fine-grained and precise way.

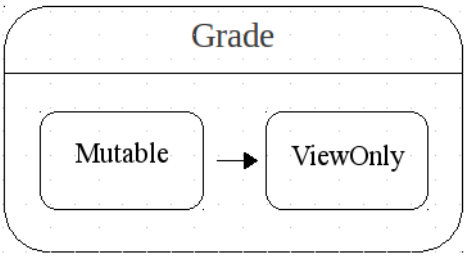
Our language uses classes to represent entities and supports first-class relationships and first-class states. We propose a language that integrates classes, relationships, and states into a single language to express access policies *declaratively* and *accurately*. By integrating the concepts of classes, relationships, and states, fined grained access control similar to ORBAC can be achieved, and the concept of state allows further refinement for access control. By using these concepts, access polices can be expressed at a high level of abstraction, facilitating reasoning about access policies by the programmer. These declarative access policies can work towards building safer mobile web applications

2. Language Proposal

Our language introduces the following abstractions: *class*, *relationship*, and *protocol*, which contains states. The following sections elaborate on these abstractions.

2.1 Class

Like classes in an object-oriented language, classes in our language are used to represent an entity, such as a `Student` or a `Professor`. Classes can be instantiated, and can contain fields and methods. In Figure 1, classes `Student`, `Class`, `Professor`, `College`, and `Dean` are declared.




```
1  protocol Grade <College co> {
2      states: Mutable, ViewOnly;
3      transitions: end(): Mutable -> ViewOnly;
4      Mutable {
5          void setGrade(Attends at, char ch) {...}
6          char viewGrade(Attends at) {...} }
7      ViewOnly {
8          char viewGrade(Attends at) {...}
9      } }
```

Figure 2. Grade state diagram and resulting Grade protocol

In addition, a class can be a *principal*. A principal represents the current user of the system.

2.2 Relationship

Our language uses first-class relationships [1, 2]. Like classes, relationships can be instantiated. In Figure 1, relationships `Attends`, `Teaches`, and `Employed` are declared. Relationships take in two arguments and associate them. Since a relationship exists between a `Class` and a `Student`, the `Class` would not need to keep references to the `Student` instances in the `Class`. We can use relationships to infer these references. With relationships, we would not need to have an id field for the `Student` class as in the ORBAC model to ensure that a Professor is teaching a particular Student. The existence of relationships also allows us to have relationship-specific fields; `grade`, for example, is a field of the `Attends` relationship.

2.3 Protocols

Protocols define states and the transitions between states. States define state-specific methods. Protocols allow us to express state machines in the code. Figure 2 shows a `Grade` state chart and the resulting `Grade` protocol. The `Grade` protocol declares two states, `Mutable` and `ViewOnly` (line 2), and one transition from `Mutable` to `ViewOnly` (line 3). A call to `end()` will trigger the transition from `Mutable` to `ViewOnly`. Methods defined in a state are accessible only in that state. The method `viewGrade()` is redundant, but the principal that is allowed to access the method differs depending on the state. As development of the language continues, ways to eliminate this redundancy will be explored.

The argument that a protocol takes in, in this case a `College` instance, signifies the instance that is affected by the state changes. The grade for each `Attends` relationship in a particular `College` can be in a `Mutable` or `ViewOnly` state.

```
1  policy writeStudentGradeByProf(Attends a,
2   Teaches t, Employed e):
3      write of a.grade
4      if principal == t.prof && t.class == a.class &&
5        t.prof == e.prof && Grade<e.col> in Mutable
```

Figure 3. Access policy for setting a grade

To facilitate re-usability, protocols can be composed with other protocols. Composition can be done in the state declaration (line 2) of a protocol.

3. Access Policies

Access policies leverage relationships, states, and the *principal*.Figure 3 shows an access policy in our language. Line 3 declares the action being permitted. Lines 4 - 5 state the requirements that need to be met to perform the action. Our policy `writeStudentGradeByProf` states that the field `a.grade` can only be written if the principal is the `prof` in the `Teaches` relationship instance `t` and if `cl` in the `Teaches` relationship instance `t` is the same as the `cl` in the `Attends` relationship instance `a` and if the `prof` in the `Teaches` relationship instance `t` is the same as the `prof` in the `Employed` relationship instance `e` and if the `Grade` protocol with argument `e.col` is in the `Mutable` state.

In ORBAC, access permissions are declared as annotations on the methods that use the field or call. In our language, access policies are declared separately from the methods and have a global effect. This ensures that different methods do not have contradicting access policies.

4. Preliminary and Expected Results

This paper describes an ongoing work.At this time, we have identified the concepts needed for our language and the constructs needed to express these concepts. We have finished a parser and will work on an interpreter and dynamic checker.

References

[1] Stephanie Balzer and Thomas R. Gross. Verifying multi-object invariants with relationships. In *ECOOP*, pages 358–382, 2011.

[2] Gavin M. Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP*, pages 262–286, 2005.

[3] Jeffrey Fischer, Daniel Marino, Rupak Majumdar, and Todd D. Millstein. Fine-grained access control with object-sensitive roles. In *ECOOP*, pages 173–194, 2009.

[4] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[5] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[6] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. In *OOPSLA*, pages 713–732, 2011.