Generation of Policy-Rich Websites From Declarative Models

by

Felix Sheng-Ho Chang

B.Sc., University of British Columbia (1999)
M.Sc., University of British Columbia (2001)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2009

Author ....
Department of Electrical Engineering and Computer Science
Jan 15, 2009

Certified by. ....
Daniel Jackson
Professor
Thesis Supervisor

Accepted by.. ....
Professor Terry P. Orlando
Chairman, Department Committee on Graduate Students

Generation of Policy-Rich Websites From Declarative Models

by

Felix Sheng-Ho Chang

Submitted to the Department of Electrical Engineering and Computer Science
on Jan 15, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Protecting sensitive data stored behind online websites is a major challenge, but existing techniques are inadequate. Automated website builders typically offer very limited options for specifying custom access policies. Manually adding access policy checks to website code is tedious and error-prone, and it is currently not feasible to automatically verify that a website conforms to its required access policy. Furthermore, policies change over time, and it can be costly to modify an existing website to reflect the changes or to certify that the modified website still complies with the desired policy.

This research presents a declarative modeling approach designed to address these issues, where the data model and the access policy are specified using Alloy, and the Weballoy tool automatically generates a dynamic website that guarantees the access policy by construction.

Thesis Supervisor: Daniel Jackson
Title: Professor

Acknowledgments

I thank my research advisor, Daniel Jackson, for his guidance and support, for encouraging me to explore different ideas and perspectives, for always being there to assist me when I needed help, and for introducing me to the world of first order relational logic and Alloy. The research opportunities and the practical impact of working with Alloy users and researchers around the world has been extremely gratifying.

I thank my thesis reader and previous research advisor at MIT, Srini Devadas, for his wise and always candid advice over the years, and for giving me the courage when I decided to purse a new area of research.

I thank my thesis reader, Butler Lampson, for his insightful feedback throughout my thesis research and for asking tough questions that both challenge and enlighten my understanding.

I thank my M.Sc. advisor at UBC, Alan J. Hu, for opening my eyes to the wondrous world of computer science research, and for giving me invaluable advice throughout my undergraduate studies, graduate studies, and beyond.

I thank my friends and lab mates at MIT — Greg Dennis, Jonathan Edwards, Eunsuk Kang, Carlos Pacheco, Derek Rayside, Robert Seater, Mana Taghdiri, Emina Torlak, and others, for their help and support during my time at MIT.

I'm grateful for my wife, Celine, for her patience and strength, and for spending more time with our children as I neared my graduation. I'm grateful for my son Edmund and daughter Denise; their miracles transformed my perspectives and my life forever. And I'm eternally grateful for my parents and my family for their love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Protecting sensitive data stored behind online websites is a major challenge. As users begin to demand more functionality from websites, many companies now allow customers to access their account information and purchase history; government agencies let citizens apply for permits, licenses, and tax claims; and wholesalers, manufacturers, and retailers often grant limited access to each other's inventory and other confidential information. Services such as Google Health[21] and Microsoft HealthVault[45] allow patients to upload their medical data online and selectively share it with specific third parties such as their doctors or hospitals. Given the personal and confidential nature of patient information, it is crucial for the access policies to be properly specified and strictly enforced. Unfortunately, existing techniques are inadequate for protecting sensitive data online.

- System and Application Access Control:

  Modern operating systems, file systems, web servers, databases, and other website middleware usually provide their own mechanism for access control. When properly layered and configured, these mechanisms can provide basic access control.

  Unfortunately, these mechanisms are usually coarse-grain and rigid: each system has a fixed idiom and cannot be easily customized. More importantly, each layer has overlapping levels of granularity and with distinct access credentials that often do not map well.

  For example, granting a newspaper reporter the permission to upload a story might involve allowing his website user account to create a new story, allowing his associated database credential to insert an entry into today's list of stories, and allowing his associated operating system account to create and modify image files on the file server. Often multiple credentials in one layer (for example, website accounts) are mapped to one or more credentials in a lower layer (such as one user id for accessing files and a different user id for serving the web pages). Credential mapping is nontrivial, and mistakes can lead to access policy violations.

- Manual Coding:

  Dynamic languages such as PHP, Perl, and Python are often used for building small to medium sized websites. Java-based frameworks such as Enterprise Java Beans, JavaServer Faces, Spring, Struts, and Hibernate offer limited automation for many aspects of website development and make Java one of the dominant language for implementing large scale enterprise websites.

  While many frameworks adopt patterns such as Model-View-Controller that make it easier to centralize and control access to data, access policy rules still have to be manually coded. Adding access policy checks to website code is tedious and error-prone, and it is currently not feasible to mechanically verify that a website conforms to its required access policy or to certify that the modified website still complies with the desired policy.

- Automated Website Builders:

  Generic website builders such as Ruby on Rails[51] and Django[9] support rapid website development by taking a high-level data model and automatically generate a skeleton website that can be further customized. Domain-specific systems such as Drupal[11], PHP-Nuke, Siebel, and Glovia produce highly-automated websites designed specifically for CMS (content management), CRM (customer relationship management), ERP (enterprise resource planning), or MRP (material requirements planning).

  These builders achieve automation by making specific idiom assumptions. As a result, they typically offer very limited options for specifying custom access policies. If a website requirement differs significantly from the tool's built-in assumptions, customizing the generated website can be difficult and the resulting website fragile, since the customization may conflict with components generated by the current or future version of the tool.

Furthermore, none of these tools provide a complete solution for long term policy maintenance: semantic difference analysis can validate that a policy change has the intended effect but cannot apply the new policy automatically, and it can be costly to modify an existing website to reflect the changes

Due to the lack of proper tool support, there have been several high-profile information leaks. In 2005, the online application and notification system used by Harvard Business School and several other business schools was found to leak the applicant's admission status from an unpublicized but obvious URL. Hundreds of applicants accessed their own admission letters a full month before the official notification date, and as a punishment, Harvard Business School rescinded the admission offers of 119 applicants [62]. Also in 2005, a web-based payroll processing website was discovered to have no access restriction on its 25,000 customers' W-2 forms, allowing any customer to access the full names, SSN, employer, and salary data of the other 25,000 customers [5]. Both incidents occurred because the access policy wasn't uniformly applied, and it wasn't feasible to mechanically verify that every web page conforms to its required access policy.

1.2 Two Key Insights

The traditional approach to building multi-user information systems and enforcing their access policies is to enumerate the list of application-level actions the user may perform in the system then define the required permissions for each action. For example, when implementing an online course management website, actions such as `AddCourse(Title, Teacher)`, `AddStudent(Course, Student)`, and `RemoveStudent(Course, Student)` would all have to be explicitly defined in at least three places: the application logic that performs the action, the security layer that enforces the access policy, and the user interface that initiates the action.

The first insight is that with a well chosen data model that matches the end user's understanding of the system, most of these actions correspond to intuitive sequences of tuple insertions and deletions from the database, so there is often no need to actually define any action in the system. There are several important consequences:

- The design is much more concise as there is no need to list the possible actions nor describe the required permission for each action. Similarly, the user interface code is much simpler as it needs only to allow users to navigate the data and insert or modify entries on screen rather than requiring a separate custom-made interaction screen for each possible action. It is even possible to automatically generate a suitable general user interface for the application from the data model alone (Chapter 4).

- The access policy rules are closer to the data model. Instead of describing two opaque actions such as `AddStudent` and `RemoveStudent`, the policy rules would describe the underlying student list and the permission required to view it, add to it, or delete from it.

  That means the policy can be validated at a much finer granularity than otherwise possible: validating that only teachers can invoke `AddStudent` does not preclude the possibility that a non-teacher can add entries to the student list via a sequence of other actions, but semantically richer rules could be verified mechanically to show that a non-teacher can never add to the student list.

  Chapter 2 describes one such style of website modeling that explicitly exposes the data model and allows the access policy to be written in terms of the data model instead of explicit actions.

- The resulting site gives more power and flexibility to its users. Instead of having predefined actions that force the users to navigate or make changes in a specific order, users can perform the changes in the order deemed more natural to them for the task at hand. For example, instead of an explicit `AddCourse` screen that asks for general course information and a second screen for adding teaching assistants to the course, the course information and the list of associated teaching assistants would all be attributes associated with a "course" entity; after creating a new course, the user can then add or remove from the attributes in any order.

The second insight is that a well chosen data model enables a separate of concerns and can be used to decouple the three main components of a multi-user information system such as a website: the database that stores the data, the security layer that guards the data, and the user interface that accesses the data:

- If data update and retrieval requests are always given in terms of objects in the data model rather than table names and row numbers, the database implementation can be replaced easily without affecting the correct behavior of the system.

- If access policies are defined based on reading, adding, and removing objects and attributes in the data model, the policy can be enforced by examining and rejecting messages going to and from the database, and there is no need to add access checks directly into various places in the main application code.

- With the database and the policy check cleanly separated, different front-end user interfaces can be used without the risk of policy violations or the need to modify the underlying system.

The simple underlying model based on tuple insertions and deletions allows automatic generation of suitable implementations for all three components since each component performs a precise task and has a well-defined dependence relationship with the other components.

As a proof-of-concept, I propose a declarative modeling approach where the website designer specifies the data model and the access policy using Alloy[1], and my tool Weballoy automatically generates a dynamic website that guarantees the access policy by construction. This framework supports rapid and incremental development, rigorous policy validation, automatic use case generation, and fully automatic website synthesis.

This thesis is organized based on the same decoupled approach: Chapter 2 presents a declarative style of modeling a website without discussing or assuming any implementation details; Chapter 3 shows how Weballoy automatically generates the database and the server-side security layer from the model; and Chapter 4 describes the automatic generation of a customizable user interface for the website.

1.3 Summary of Contributions

The work presented in this thesis makes the following contributions to website specification, validation, and synthesis research:

- A new framework for access policy specification and an architecture for mandatory enforcement.

- A novel specification language, embedded in Alloy, for concisely expressing the data model and access policy of a website (Chapter 2).

- A website synthesis tool Weballoy that embodies the approach described in this thesis and guarantees the access policy by construction (Chapter 3).
- An Alloy-aware template engine and language for safe customization of the generated website (Chapter 4).

Case studies completed to date include the automatic synthesis of a collaborative online website for the Alloy research group, a social networking website modeled after MIT FamilyNet[13], and a conference management website modeled after Continue[40]. These are described in detail in Chapter 5.

Chapter 2

Website Modeling

This chapter briefly introduces the Alloy modeling language and describes how it can be used to model the database contents, consistency constraints, and access policies for a dynamic website. This text presents key features of Alloy by using an online grades management website as a running example, and does not assume familiarity with the language, though readers are encouraged to consult the Alloy Language Reference [1].

2.1 Data Model

2.1.1 Signatures

The first step in modeling a dynamic website using Weballoy is to express the various entities and their relationships as a data model in Alloy.

Given the data model, Weballoy automatically generates the corresponding database schema and consistency constraints. The Alloy model does not explicitly model the notion of time steps or state transitions; instead, successive database states correspond to a sequence of satisfying instances for the Alloy data model.

Each *signature* in Alloy represents a set of objects known as *atoms*. In addition to Alloy's standard signatures `Int`, `Bool`, and `String`, users can also import the Weballoy library module (shown in Figure 2-2) which provides additional signatures such as `Date`, `DateTime`, `Email`, and `Password`.

Example. An initial data model for an online grades management website might start with the following:

```
abstract sig User { }
sig Teacher, Student extends User { }
sig Course { }
```

The `User` signature is designated as *abstract*, meaning there cannot be a User who is neither a Teacher nor a Student.

2.1.2 Fields

Each relationship can be represented by a *field* in a signature. Each field is a binary (or higher arity) relation from a signature to another set or relation. Furthermore, each field can be constrained with *multiplicity* markers such as `some` (at least one), `lone` (zero or one), or `set` (meaning the size of this field is unconstrained).

Example. To model the notion of people teaching and attending courses, the previous Alloy model might be augmented as follows:

```
abstract sig User {
  name: String,
}
sig Teacher extends User {
  teaches: set Course
}
sig Student extends User {
  assists, attends: set Course
}
sig Course {
  name: String,
  teachers: set Teacher,
  assistants, students: set Student,
  grades: students -> lone Int
}
```

Each field is a relation from its containing signature to the entities listed in the field declaration: the `teaches` field is a binary relation from `Teacher` to `Course`, and the `grades` field is a ternary relation `Course->Student->Int` where each student in a course is associated with at most one integer.

2.1.3 Metamodel

Every Alloy model corresponds to an instance of the Alloy *metamodel*, where each signature `s` in the original model corresponds to a *meta atom* denoted `s$`, and each field `f` in the signature `s` corresponds to a *meta atom* denoted `s$f`.

Given a meta atom `m` corresponding to a signature, `m.fields` denotes the set of meta atoms corresponding to the fields of that signature, and `m.value` denotes the elements in that signature. Likewise, given a meta atom `m` corresponding to a field, `m.value` denotes the tuples in that field.

Allowed behaviors of a model can be specified reflectively by reasoning over the metamodel, similar to the way languages such as Java provide a reflection mechanism for a program to refer to its own code and data structures. When specified in particular formats recognized by Weballoy as described in Section 2.3, the behavior specification can serve dual purposes: it can be validated for correctness by the Alloy Analyzer, and it can be converted into access policy checks that will be automatically enforced by the generated websites.

Example. The following predicate evaluates to true when `c1` and `c2` have the same values for every field.

```
pred equal [c1, c2: Course] {
  c1.name        = c2.name
  c1.teachers    = c2.teachers
  c1.assistants  = c2.assistants
  c1.students    = c2.students
  c1.grades      = c2.grades
}
```

This can be written more succinctly by quantifying over the fields of the `Course` signature:

```
pred equal [c1, c2: Course] {
  all f: Course$.fields | c1.(f.value) = c2.(f.value)
}
```

The expression `Course$.fields` denotes the set of meta atoms corresponding to fields in the `Course` sig. For each meta atom in `Course$.fields`, the function `value` returns the actual field that the meta atom corresponds to. Therefore, this predicate says `c1` and `c2` have the same values for every field.

2.1.4 Facts

Similar to the notion of consistency constraints in a relational database, an Alloy model can contain *facts* which must be maintained at all times.

Example. The following fact enforces the constraint that the `teachers` and `teaches` relations are exactly inverses of each other:

```
fact { teachers = ~ teaches }
```

This fact requires both fields to be updated together. If this is undesirable, expansion triggers (explained later in Section 2.3.3) can be used to automatically update one field when the other field is modified.

2.2 Semantic Model

2.2.1 Transactions

In keeping with Alloy's relational semantics, users request changes to the database by adding atoms to a signature, destroying atoms, adding tuples to a field, and removing tuples from a field.

Example. Continuing from the previous example, changing the name of `course` `c1` from "Networking" to "Advanced Networking" requires the removal of tuple (`c1`→`"Networking"`) from the `name` field followed by the addition of tuple (`c1`→`"Advanced Networking"`) to the `name` field.

Modification requests are grouped into *transactions* where each *transaction* would either be performed in its entirety or be rejected as a whole. For simplicity, the current semantics guarantee that the transactions are serializable, though this may be relaxed in future versions of the tool.

Each transaction is a pair of sets (AS, DS) where AS is the set of atoms and tuples to be added, and DS is the set of atoms and tuples to be removed.

For each (m, x) pair in AS, either m is a meta atom denoting a signature and x is a fresh atom to be added to that signature, or m is a meta atom denoting a field and x is a new tuple to be added to that field.

Likewise, each (m, x) pair in DS represents either a signature m and an atom to be deleted from that signature, or a field m and a tuple to be removed from that field.

Example. Suppose teacher `t1` logs in to the website and wishes to remove an old course `c1` and create a new course `c2`, he might submit the following transaction:

```
({
   (Course$, c2),                                  ⎫
   (Course$name, c2→"Software Engineering"),       ⎬ AS
   (Course$teachers, c2→t1)                        ⎭
}, {
   (Course$, c1)                                   ⎫
})                                                 ⎭ DS
```

When the atom `c1` is destroyed, any tuple containing `c1` is also removed. The user does not need to explicitly remove the tuple (`c1` → `"Advanced Networking"`) previously added to the `name` field.

If a transaction attempts to create and destroy the same atom, or attempts to add and remove the same tuple from the same field, the transaction is rejected as malformed. Therefore, the order of the requests in the transaction doesn't matter.

2.2.2 Queries

Users retrieve tuples from the database by issuing queries. Each query RS is a set of pairs (x, f) where each x is an atom and each f is a meta atom corresponding to a field of x that the user wants to retrieve.

Example. The following query retrieves the set of courses taught by `t1`, followed by the set of teachers teaching `c1`:

```
{
   (t1, Teacher$teaches), (c1, Course$teachers)
}
```

Each query is handled atomically: no transaction is applied during the processing of a query, thus the query result always corresponds to a consistent database state.

2.3 Access Policy

This section describes a style of declarative policy specification where the policy will be meaningful and checkable using the Alloy Analyzer and yet have a specific operational semantics when used to drive the automatic website synthesis. When specified according to the conventions described below, the access policy can be mechanically validated by the Alloy Analyzer (Figure 2-1), as well as automatically enforced by the generated website (Figure 3-2).

The key trick here is to use specially named *predicates* and *functions* where the type of argument associated with the predicate or function is used to associate it with a particular signature or field.

The bodies of these predicates and functions can refer to a singleton set named `me` containing the current logged-in user's atom (or the empty set if the user is accessing the website without logging in).

Currently, Weballoy recognizes four types of special constructs: pre-conditions, post-conditions, expansion triggers, and capabilities.

2.3.1 Pre-conditions

Pre-conditions are predicates that are evaluated before an atom is created or destroyed, or before a tuple is inserted or removed. If the predicate evaluates to false, the transaction is rejected.

Pre-conditions on creating an atom for a given signature `s` can be specified by declaring a predicate `s.preAdd` in the model. Pre-conditions on adding a tuple to a given field `f` can be specified by declaring a predicate `f.preAdd`.

Example. Continuing from the previous example, the following pre-condition on the `assistants` field states that a user cannot add teaching assistants to a course if the user is not teaching that course:

```
pred assistants.preAdd{
  (one me) and (me in this.first.teachers)
}
```

This declares a predicate `preAdd` with a single parameter. The parameter is named `this` by default, and it is declared to be a subset of the `assistants` field, so an actual value of this parameter will be a pair consisting of a Course and a Student. If a user tries to add `s1` to `c1.assistants`, the pre-condition `preAdd[c1->s1]` must be satisfied. The helper function `first` (shown in Figure 2-2) returns the first atom from the tuple. The function call `this.first` returns `c1`, and `this.first.teachers` gives us the teachers of that course. If the user is not logged in, `me` evaluates to the empty set which is always a subset of any other set. Therefore, the `one` keyword is needed to ensure the user is logged in as one of the teachers.

Likewise, pre-conditions on atom destruction and tuple removal can be specified by declaring predicates `s.preDel` and `f.preDel` respectively.

Example. The following pre-condition states that a user cannot remove a teaching assistant from a course if the user is not the teacher nor the teaching assistant being removed:

```
pred assistants.preDel {
  (one me) and (me in this.first.teachers+this.second)
}
```

If a user attempts to remove `s1` from `c1.assistants`, the pre-condition `preDel[c1→s1]` must be satisfied. The expression `this.second` gives us the student `s1`. Thus the request will be denied if the user is not `s1` nor one of the teachers of `c1`.

2.3.2 Post-conditions

Post-conditions are predicates that must evaluate to true after atoms are created or destroyed, or after tuples are inserted or deleted. In particular, post-conditions on atom creation are often used to specify the legal initial field values of an atom. If any post-condition evaluates to false, changes made so far are rolled back, and the entire transaction is rejected.

Post-conditions on creating an atom for signature `s` or adding a tuple to field `f` can be specified by declaring a predicate `s.postAdd` or `f.postAdd` respectively. These post-conditions are evaluated after all atom creations and tuple insertions specified in this transaction have occurred but before any atom destruction or tuple deletion.

Example. The following post-condition ensures that a newly created course has exactly one teacher (equal to the creator of the course) and no assistants nor students:

```
pred Course.postAdd {
   this.teachers = me
   one this.teachers
   no this.assistants + this.students
}
```

Likewise, post-conditions on destroying an atom of signature `s` or removing a tuple from field `f` can be specified by declaring a predicate `s.postDel` or `f.postDel` respectively. These post-conditions are evaluated after the entire transaction has been performed.

Example. The following post-condition on the `teachers` field ensures the last teacher of a course cannot be removed:

```
pred teachers.postDel { some this.first.teachers }
```

2.3.3 Expansion triggers

Expansion triggers enable changes in one field to automatically propagate to other fields. Every tuple insertion and deletion can trigger a reaction which results in more tuple insertions or deletions. To simplify the semantics, expansion triggers are required to be monotonic: tuple insertions can trigger additional tuple insertions but not tuple deletions, and tuple deletions can trigger additional tuple deletions but not tuple insertions.

Given fields `f` and `g`, the user can declare a function "`f.onAdd: g`" to denote the set of tuples that should be automatically added to `g` whenever `f` grows. Likewise, "`f.onDel: g`" denotes the set of tuples that should be automatically removed from `g` whenever `f` shrinks.

Example. Continuing from the previous example, the consistency constraint "`teachers = ~ teaches`" shown earlier is inconvenient for users because it rejects a transaction if the user does not make the corresponding changes to both fields in the same transaction. To remedy this, the following expansion trigger definitions can be added to automatically propagate changes in one field to the other:

```
fun teachers.onAdd: teaches { this.second -> this.first }
fun teachers.onDel: teaches { this.second -> this.first }
fun teaches.onAdd: teachers { this.second -> this.first }
fun teaches.onDel: teachers { this.second -> this.first }
```

When teacher `t1` is added to course `c1`'s list of teachers, the tuple `c1->t1` is passed to the `teachers.onAdd` function. In this case, it returns exactly one tuple: `t1->c1` which is automatically added to the `teaches` field. This new tuple also triggered the `teaches.onAdd` expansion trigger which determines that the original tuple `c1->t1` should be added to the `teachers` field. Since `c1->t1` is already present, no further action is triggered, and the expansion processing is complete.

Expansion triggers apply after pre-conditions have been evaluated, so these extra tuple insertions and removals are exempt from the pre-condition checks but not the post-condition checks. If a trigger-induced tuple insertion or removal violated a post-condition, the transaction will be rejected.

Expansion triggers can also propagate changes between a field and a signature, or between two signatures. Given signature `A`, signature `B`, and field `f`:

"`A.onAdd: f`" denotes the set of tuples to add to `f` whenever `A` grows.
"`A.onDel: f`" denotes the set of tuples to remove from `f` whenever `A` shrinks.
"`f.onDel: A`" denotes the set of atoms to delete from `A` whenever `f` shrinks.
"`A.onDel: B`" denotes the set of atoms to delete from `B` whenever `A` shrinks.

The other two combinations "`f.onAdd: A`" and "`A.onAdd: B`" are not currently supported, since the Alloy language does not have constructors so there is no way for an expansion trigger to refer to an atom that does not exist yet.

2.3.4 Capabilities

If many fields have the same access policy, it is tedious to define the required permission for each field using a separate pre-condition rule. To remedy this, a separate access policy mechanism based on *capabilities* is provided. Each *capability* is a triple of type `Action->univ->univ`.

There are three `Action` atoms: `R` (for retrieval), `A` (for addition), and `D` (for deletion), and there are five types of capabilities: atom creation, atom destruction, tuple insertion, tuple removal, and tuple retrieval.

(1) Creating a new atom in signature s requires the capability triple (`A, A, s$`).

(2) Destroying an existing atom x requires the capability triple (`D, D,` x).

(3) Inserting the tuple (x_1, x_2, …, x_n) into field `f` in signature `s` requires the capability triple (`A`, x_1, `s$f`).

(4) Removing the tuple (x_1, x_2, …, x_n) from field `f` in signature `s` requires the capability triple (`D`, x_1, `s$f`).

(5) Reading the tuple (x_1, x_2, …, x_n) from field `f` in signature `s` requires the capability triple (`R`, x_1, `s$f`). These `R` capabilities are ignored when processing a transaction; instead, they determine whether a query is permitted or not as described later in Section 2.3.6.

For convenience, the Weballoy library module (shown in Figure 2-2) predefines four useful functions: `ADD = A->A`, `DELETE = D->D`, `W = A+D`, and `RW = R+W`.

To exploit this mechanism, the model is augmented with a function named `policy` that takes no arguments and returns a ternary value of type `Action->univ->univ`. If no such function is defined in the model, then the capability checks are not performed, and queries and transactions are approved by default unless rejected by a consistency constraint or a pre- or post-condition.

Example. Continuing from the previous example, suppose teachers of a course should be able to modify every field of that course. Instead of writing 10 pre-conditions (5 `preAdd` and 5 `preDel` predicates), this can be expressed more succinctly by adding the following function to the model:

```
fun policy: Action -> univ -> univ {
  W -> me.teaches -> Course$.fields
}
```

`W` is a convenient function provided by the Weballoy module (Figure 2-2) that returns the union of `A` and `D`. The expression `me.teaches` evaluates to the set of all courses being taught by the user making the request, and `Course$.fields` denotes the meta atoms corresponding to the fields of `Course`. So this says that teachers of a course can modify every field if not prohibited by other pre- or post-conditions or a consistency constraint.

Whenever a request is made, the generated website associates the singleton set named `me` with the atom corresponding to the user making the request (or the empty set if the user is accessing the website without logging in), and evaluates the `policy` function to compute the set of capabilities currently possessed by the user. If the user does not possess all required capabilities for this transaction, the transaction is rejected.

> *Example.* Suppose teacher `t1` wishes to remove `s1` from `c1.assistants` where the current database is as follows:
>
> ```
> Teacher = t1+t2+t3.   c1.assistants = s3.   t1.teaches = c1+c2.
> Student = s1+s2+s3.   c2.assistants = s1.   t2.teaches = c3.
> Course  = c1+c2+c3.   c3.assistants = s2.   t3.teaches = c1.
> ```
>
> The capabilities possessed by `t1` are determined by evaluating the `policy` function with `me` = `t1`. The expression `me.teaches` evaluates to {`c1, c2`}, so `t1` currently has 20 capability triples resulting from the cross product {`A, D`} → {`c1, c2`} → {`Course$name, Course$teachers, Course$assistants, Course$students, Course$grades`}. The required capability `D` → `c1` → `Course$assistants` is one of the 20 capabilities currently possessed by `t1`, so the request is approved if it does not violate another constraint such as a pre- or post-condition.

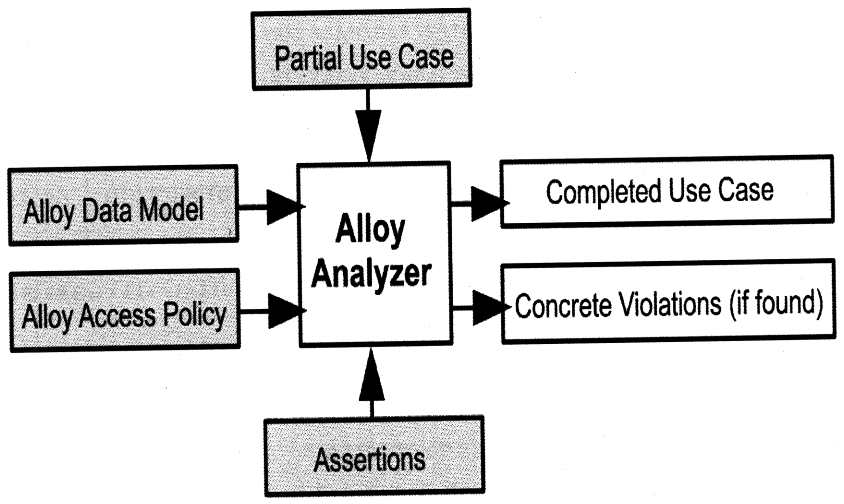
Compared with typical role-based access control systems, this approach is much more flexible. Instead of granting permissions to roles that have to be explicitly created and assigned by administrators, the `policy` function does not require an explicit notion of roles and can assign state-dependent permissions to users at run time.

2.3.5 Processing a Transaction

The pre-conditions, post-conditions, expansion triggers, and capabilities are checked in the following order:

- When a transaction is received, the website first checks the capability rules to make sure the user has the required capability for every request in the transaction.
- If the user has the required capabilities, the system checks the pre-conditions for atom creation and tuple insertion requests, applies the `onAdd` triggers perhaps repeatedly (as shown in Figure 2-3), performs the creations and insertions, then checks their post-conditions.
- If any pre-condition or post-condition is violated, the entire transaction is rejected. Otherwise, the system checks the pre-conditions on atom destruction and tuple removal requests, applies the `onDel` triggers, performs the destruction and removals, then checks their post-conditions.
- Finally, if any *fact* is violated, all changes are rolled back and the transaction is rejected.

Given the set of insertions AS and the set of deletions DS as described in Section 2.2.1, the pseudocode for processing the incoming transaction is shown in Figure 2-3.



(Gray boxes denote required manual inputs)

Figure 2-1: Policy Validation Workflow

As mentioned previously, atom creations and tuples insertions happen before atom destructions and tuple removals. This ensures expansion triggers are monotonic: when adding tuples, the corresponding `onAdd` triggers are evaluated to expand the set of tuples to be inserted until there are no more new tuples to insert. Later when removing tuples, the corresponding `onDel` triggers are evaluated to expand the set of tuples to be removed until there are no more tuples to remove.

This has two important consequences: first of all, it means the behavior of the system is deterministic and does not depend on which enabled trigger is applied first. Secondly, since the trigger functions are evaluated until a fixed point is reached before performing the actual insertion or removal, trigger evaluations of a particular transaction are all done on the same database pre-state and may be partially parallelized.

2.3.6 Processing a Query

When a query is received, the website does not need to evaluate any of the pre-conditions, post-conditions, or consistency constraints since queries cannot modify the database. Instead, the website associates the singleton set named `me` with the atom corresponding to the user making the query (or the empty set if the user is not logged in) then evaluates the capability function `policy` to determine the capabilities currently possessed by the user.

As previously mentioned in Section 2.3.4, reading the tuple (x_1, x_2, ..., x_n) from field `f` in signature `s` requires the capability triple (`R`, x_1, `s$f`). If the user does not possess all capabilities required by a query, the entire query is denied. Otherwise, the tuples requested in the query are returned to the user.

The pseudocode for processing an incoming query is shown in Figure 2-3.

2.4 Validation

Given a data model and the associated access policy expressed in Alloy, the Alloy Analyzer can be used to validate whether the model accurately represents the website designer's intentions (Figure 2-1). The Alloy `run` command can be used to automatically generate use cases from partial use case descriptions, whereas the Alloy `check` command performs bounded model checking to verify whether the model satisfies user-provided properties.

Example. Suppose the website designer wishes to subsume the `assistants.preAdd` pre-condition (shown in Section 2.3.1) with the capability rule shown in Section 2.3.4. The following `check` command can be used to check whether the pre-condition for adding an assistant to a course is true if and only if the current user possesses the corresponding capability triple for adding assistants:

```
check {
  all c:Course, s:Student |
    preAdd[c->s] iff A->c->Course$assistants in policy
}
```

2.5 Related Work

2.5.1 Access Control Specification Languages

Domain-specific access control specification languages exist for a variety of environments such as operating systems, network firewall configurations, relational databases, and enterprise resource management. While these languages have been successful, they are often very specialized and have limited tools support since tools written for one language cannot be used with another.

Many general purpose access control systems are now based on the principle of Role-based Access Control (RBAC)[14]. RBAC improves upon traditional Mandatory Access Controls [56] by allowing administrators to create specific roles authorized for different tasks. Each user can be assigned to zero or more roles. Instead of controlling access to low-level objects, RBAC allows operations to be grouped into transactions with specific meaning and access policy in a particular organization. Furthermore, the roles may be dynamically reassigned without changing the policy itself.

Increasingly, RBAC-based languages such as XACML[58] are being proposed as a uniform approach to specify the access policy for heterogeneous resources. Compared with Alloy, XACML has more built-in primitives for conveniently describing certain policies, but cannot express complex conditions involving universal or existential quantifiers. For example, when modeling the "Continue[40]" conference management system (Section 5.2), it is impossible to express the following constraint using XACML: "conference chairs cannot move a conference into the discussion phase if there exists at least one unreviewed paper."

2.5.2 Policy Validation

Many tools exist for validating the correctness of role-based access control policies. Margrave[16] analyzes policies given in XACML and can compare the semantic difference between two policies. Becker et al.[3] translates authorization rules to Transaction Logic and uses Datalog to verify properties of the policy. Haidar et al.[23] models the role-based access policy of a conference management website by explicitly listing all possible events and using Z[52] to specify when each event is allowed to occur. Flores et al.[18] represents the legal sequences of web page visits as a Kripke[39] structure and uses an LTL[49] model checker to validate semantic properties.

Unlike the Alloy Analyzer which performs a straightforward bounded encoding of arbitrary assertions into SAT, these tools were designed specifically for analyzing access policies and are often more efficient or can check unbounded properties. On the other hand, the access policy languages used by these tools do not support automatic code generation.

2.5.3 Data Models

Many textual and graphical notations such as Data Structure Diagrams and Entity-Relationship Diagrams were invented to describe the data structure of a computer system. More recently, UML[26] has been standardized by the Object Management Group as a suite of general purpose notations for describing both the data model and the behavior of a system, and OCL[29] complements UML's graphical notations by allowing users to specify additional constraints in a precise and textual format.

UML has been widely adopted in the computer software industry and is supported by a variety of commercial tools. However, UML is much more complicated than Alloy. Despite various attempts to precisely define UML, major UML tool-chains often do not interoperate well due to differences in their interpretation of the standard. Moreover, even though UML and OCL are very expressive, the complexity of the notation may encourage users to write human language annotations instead of precise UML markings, further reducing the effectiveness of automation offered by these tools. In contrast, Alloy is much simpler, as it is based on a very straightforward first-order logic; and Alloy semantics is precise, therefore allowing fully automatic test case generation, property checking, and (for certain domains such as web site construction described in this thesis) automatic code generation.

2.5.4 Object-Relational Mapping (O/R-M)

O/R-M addresses the impedance mismatch between an object-oriented programming language and an SQL relational database that does not support rich data structures. By associating primitive values in database tables with live objects in programs at run time, the programmer does not need to access the database contents explicitly; instead, the contents become part of the program's state space as changes to the objects propagate to the database and vice versa.

O/R-M is widely implemented in many tools and frameworks. Hibernate[24] is a Java framework allowing the mapping to be specified in either direction: Java code can be used to derive a suitable database schema, and an existing database schema can be used to generate skeleton Java code. Ruby on Rails[51] generates skeleton Ruby code from a data model and allows database contents to be accessed transparently if desired. Magritte[50] also supports object-relational mapping and offers a meta-model facility in Smalltalk for displaying, validating, and modifying the data objects.

Alloy atoms and signatures are similar to instances and classes in an object-oriented languages, so the transparent schema generation and database access described in Section 2.1 roughly corresponds to performing object-relational mapping between the database and an Alloy instance. However, existing O/R-M frameworks do not support rich access policy beyond the primitive mechanisms implemented in the underlying SQL database.

```
module Weballoy

sig Date       = Int    { }
sig DateTime   = Int    { }
sig Email      = String { }
sig Password   = String { }
sig LongString = String { }

lone sig me in univ { }

abstract sig Entity {
  name:     lone String,
  owners:   set Entity,
  created:  DateTime,
  modified: DateTime,
}

abstract sig NamedEntity extends Entity { } {
  some name
}

abstract sig LoginUser extends NamedEntity {
  suspended: Bool,
  email:     disjoint Email,
  password:  Password
}

enum Action { R, A, D }
fun W:      Action { A + D }
fun RW:     Action { R + W }
fun ADD:    Action->Action { A -> A }
fun DELETE: Action->Action { D -> D }

fun first  [x: univ->univ] : univ { x.univ }
fun second [x: univ->univ] : univ { univ.x }
```

Figure 2-2: The Weballoy library module

Pseudocode for processing a query *RS*:

```
$\forall$ (atom $x$, field $m$) $\in RS$, reject if (R -> $x$ -> $m$) $\notin$ policy
Otherwise, perform the query and give the answers back to the user.
```

Pseudocode for processing a transaction (*AS*, *DS*):

Step 1: Capability check

```
$\forall$ (signature $m$, atom $x$) $\in AS$:
reject if $t$ is not fresh or if (A -> A -> $m$) $\notin$ policy

$\forall$ (signature $m$, atom $x$) $\in DS$:
reject if $t$ is not an existing atom or if (D -> D -> $x$) $\notin$ policy

$\forall$ (field $m$, tuple $(x_1, x_2, ..., x_n)$) $\in AS$:
reject if $x_1$ is not fresh and (A -> $x_1$ -> $m$) $\notin$ policy

$\forall$ (field $m$, tuple $(x_1, x_2, ..., x_n)$) $\in DS$:
reject if $x_1$ is not an existing atom, or if (D -> $x_1$ -> $m$) $\notin$ policy
```

Step 2: Validate the atom creation and tuple insertions

```
$\forall$ ($m$, $t$) $\in AS$ | $\forall$ predicate $p$ named "$m$.preAdd" | reject if $\neg$ $p[t]$.
let $Q = \emptyset$.
while ($AS \neq Q$) {
    choose ($f$, $t$) from $AS - Q$.
    add ($f$, $t$) to $Q$.
    $\forall$ function $p$ named "$f$.onAdd: $g$" | add $(g, p[t])$ to $AS$.
}
Add atoms and tuples in $AS$ to the database.
$\forall$ ($m$, $t$)$\in AS$ | $\forall$ predicate $p$ named "$m$.postAdd" | reject if $\neg$ $p[t]$.
```

Step 3: Validate the atom deletions and tuple removals

```
$\forall$ ($m$, $t$) $\in DS$ | $\forall$ predicate $p$ named "$m$.preDel" | reject if $\neg$ $p[t]$.
let $Q = \emptyset$.
while ($DS \neq Q$) {
    choose ($f$, $t$) from $DS - Q$.
    add ($f$, $t$) to $Q$.
    $\forall$ function $p$ named "$f$.onDel: $g$" | add $(g, p[t])$ to $DS$.
}
Delete atoms and tuples in $DS$ from the database.
$\forall$ ($m$, $t$)$\in DS$ | $\forall$ predicate $p$ named "$m$.postDel" | reject if $\neg$ $p[t]$.
```

Step 4: Consistency check

```
If every fact defined in the model is still true, commit the changes;
otherwise roll back all changes and reject the request.
```

Figure 2-3: Query and transaction processing

Chapter 3

Back-End Synthesis and Mandatory Policy Enforcement

3.1 Overview

The preceding chapter presented a declarative style of modeling where the data model and the access policies of a website are specified and reasoned about using Alloy. It showed how to describe the expected behavior of the corresponding website but did not explain how to match a model with a particular website. One way to ensure that the website obeys the access policy is to synthesize the entire website automatically using a code generator. This chapter describes my implementation of a code generator, Weballoy, that enforces the access policy by construction.

The architecture for websites generated by Weballoy is shown in Figure 3-1. JavaScript code running in the web browser displays a web page by issuing queries to Java servlets running on the server. If the query is approved, the servlet will perform the query and return the results back to the JavaScript code which displays it for the user.

Likewise, when the user makes changes on the website, the JavaScript code combines the set of additions and deletions indicated by the user into a transaction and sends it to Java servlets running on the server. The transaction undergoes the four stages of access policy check mentioned in Section 2.3.5 and shown in Figure 2-3. If any stage rejects the transaction, the servlet returns an error message to the user, otherwise the entire transaction is committed to the database.

The database schemata, Java servlets, HTML, and JavaScript code are all automatically generated by Weballoy. The only required inputs are the Alloy data model and Alloy access policy specification. The website synthesis workflow is shown in Figure 3-2.

The front-end JavaScript and the back-end server code are decoupled and communicate by sending Remote Procedure Call (RPC) messages via HTTP POST. The five message types currently supported are described in Section 3.6. By choosing industry standard formats and protocols, the front-end and back-end are safely decoupled, providing two important benefits.

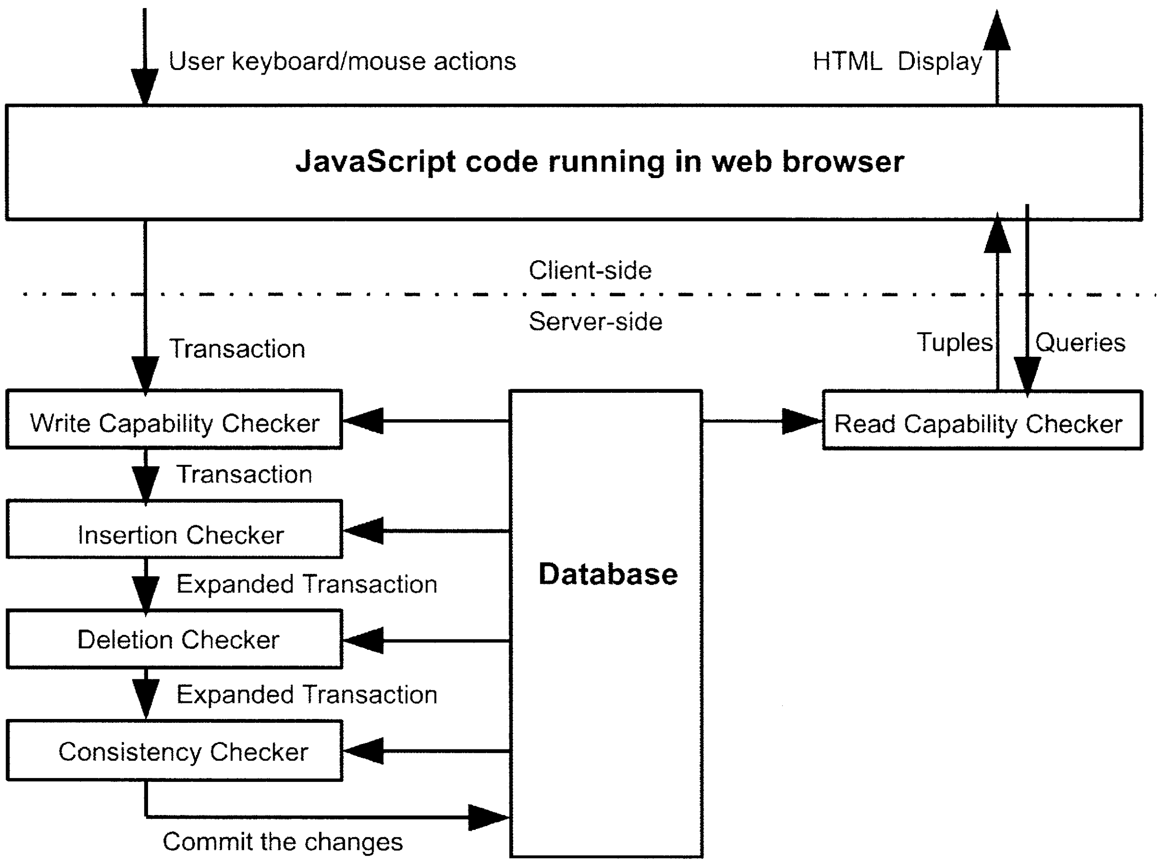


Figure 3-1: Architecture of Synthesized Website

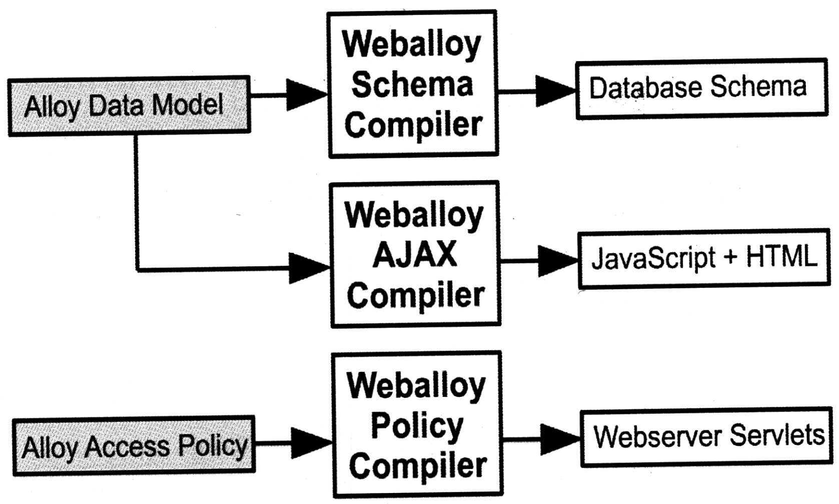
First, the same back-end that services web browser requests can also act as a web service provider. This allows easier integration of the data store with other enterprise business applications, and provides a single gatekeeper where the access policy can be enforced for every application that accesses the data.

Second, the front-end web page display is cleanly separated and can be replaced and customized for each website as needed. Chapter 4 describes a technique for automatically generating interactive JavaScript code and a new Alloy-aware template language for customizing the appearance, but existing industry template engines such as XSLT, JSF, PHPTemplate, or even manual coding could be used instead without the possibility of unintended policy violations.

Next I'll describe the components shown in Figure 3-1: the database, the servlets responsible for policy checks, the client-side JavaScript code, and the messaging protocols between JavaScript and the server.

3.2 Database

The current Weballoy implementation does not sit on top of an off-the-shelf SQL relational database since it is nontrivial to translate Alloy expressions into efficient SQL queries. Instead, a new relational tuple store was implemented that natively supports queries containing Alloy relational operators.



(Gray boxes denote required manual inputs)

Figure 3-2: Website Synthesis Workflow

The database contents are stored as an indexed file on disk, where each signature is associated with its current set of atoms and each field is associated with its current set of tuples. Atoms and tuples are fetched on demand. When atoms are read in to main memory, they are interned immediately. This allows atom equality to be determined efficiently by direct pointer comparison.

Each tuple is represented by a pointer to an array of atoms. Since most tuples are discarded from a query, it is wasteful to spend time interning them. For example, when evaluating the relational join between a binary relation `f` and an atom `x`, tuples in `f` that do not end in `x` are all discarded.

Instead, when two tuples are compared, hash values based on the atoms in the tuples are computed and memorized. If the hash values are different, they are necessarily different tuples and the inequality is confirmed immediately. If the hash values are the same, they are compared atom-by-atom. If found to be equal, one tuple object will be modified to point to the same array of atoms as the other tuple object, so the next time these two tuples are compared, they will be deemed equal immediately since they point to the same underlying array of atoms.

3.3 Evaluator

The evaluator is an integrated part of the custom tuple store responsible for converting arbitrary Alloy expressions into primitive query operations supported by the tuple store.

When optimizations are not enabled, the evaluator works by fetching the contents of every signature and every field mentioned in an Alloy expression into main memory then performing the corresponding relational operations on them. After conducting the first few case studies, several important optimizations were added to the evaluator, including *signature elision* and *membership decomposition.*

```
E .  S = { a         | a->b in E for some atom b of type S }
S .  E = { b         | a->b in E for some atom a of type S }
E &  S = { a in E    | a is an atom of type S              }
E –  S = { a in E    | a is an atom whose type is not S    }
S <: E = { a->b in E | a is an atom of type S              }
E :> S = { a->b in E | b is an atom of type S              }
```

Figure 3-3: Important signature elision rules (given expression `E` and signature `S`)

Given expressions E_1 and E_2, signature `S`, and atoms $x_1, \cdots, x_n$:
(1) x_1 `in S` if x_1 is an atom of type `S`
(2) E_1 `in` E_2 if every tuple in E_1 is also in E_2
(3) $(x_1, \cdots, x_n)$ `in` E_1`+`E_2 if $(x_1, \cdots, x_n)$ `in` E_1 or $(x_1, \cdots, x_n)$ `in` E_2
(4) $(x_1, \cdots, x_n)$ `in` E_1`->`E_2 if $(x_1, \cdots, x_m)$ `in` E_1 and $(x_{m+1}, \cdots, x_n)$ `in` E_2
where E_1's arity is m and E_2's arity is $n - m$.

Figure 3-4: Important membership decomposition optimization rules

Signature elision refers to a collection of rules (Figure 3-3) intended to avoid retrieving the full contents of signatures. The intuition is that a formula often contains references to signatures in the model but the truth value of the formula does not depend on the entire signature.

For example, if `s` is a signature, then the expression `(x & s)` can be efficiently evaluated by computing `x` then removing all atoms whose type is not `s`; empirically, `x`'s size is often a small number whereas signature `s` can be of arbitrary size as more data are added to the database.

Membership decomposition (Figure 3-4) avoids explicit construction of large relations corresponding to expressions in membership tests by converting unions into disjunctions and products into conjunctions. For example, given sets `X`, `Y`, and `Z`, the membership test `(X->Y->Z in A->B->C)` is simplified to `(X in A and Y in B and Z in C)`, and `(X in A + B + C)` is rewritten as `(all x: X | x in A or x in B or x in C)`. Once decomposed, membership tests between disjoint sets can be simplified to false without evaluating the arguments.

Membership formulas of this form arise very naturally when performing the capability check, where the left hand side is often small since it corresponds to a user initiated transaction, but the right hand side is large since it contains cross products of every entity in the database. Combined with the *signature elision* optimization mentioned earlier, capability checks can often be performed without ever retrieving the full content of any signature from the database.

3.4 Java Servlets

The servlets running on the web server are responsible for processing incoming queries and transactions. Since the tuple store natively supports Alloy expressions and formulas, the servlets simply implement the pseudocode shown in Figure 2-3.

At run time, the servlets invoke the evaluator to evaluate the `policy`, `onAdd`, and `onDel` functions, and the `preAdd`, `preDel`, `postAdd`, and `postDel` predicates as needed. No additional optimizations are implemented in the servlets themselves; instead, major bottlenecks are identified and improved in the evaluator. The same evaluator is also used in the Alloy Analyzer allowing users to interactively evaluate expressions in an instance, so the speed-up benefits all users of the evaluator and not just the servlets.

There are five different RPC message types for handling incoming queries and transactions. The wire format is plain text JSON (JavaScript Object Notation), and the current implementation uses the Google Web Toolkit [22]'s GWT-RPC library for making the remote procedure calls and receiving responses from call-backs. These five methods are described in Section 3.6 using GWT-RPC notation, though clients are free to issue requests using other compatible libraries.

3.5 Session Cookies

All GWT-RPC calls are done by making HTTP requests, so the client's HTTP cookie is an implicit argument that is always passed via the "HTTP Cookie" header during each RPC call. As is standard in most frameworks, the cookie is used to identify the requests as belonging to a particular user, and this association is used during access policy checks for determining whether to grant the request. To do this, the servlets maintain a mapping from each valid cookie to the atom of the user who currently possesses that cookie.

When a client issues a request without sending a cookie, or with a cookie that has expired, the server generates a random but currently unused cookie and issues it to the client via the "HTTP Set-Cookie" reply header. Web browsers typically honor the Set-Cookie reply and will send this new cookie along in all subsequent messages to the server. But refusal to accept the new cookie is not a policy violation risk: at this point the cookie is not yet associated with any particular user so the client still has the same permission as any anonymous visitor.. In particular, policy checks (as described in Section 2.3.5 and 2.3.6) for requests coming from this client will be evaluated by associating the specially named value `me` with the empty set.

If the user logs in with a valid password, a fresh cookie associated with the user's atom will be issued to the user via the "Set-Cookie" reply header. Subsequent access by the user will be checked by associating `me` with his atom rather than an empty set, until either the cookie expired due to inactivity or the cookie is invalidated explicitly by calling `logout()`.

3.6 RPC Message Types

As mentioned earlier, the generated server code currently provides five callable RPC methods. They are shown in Table 3.1.

Method	Description
`login`	Returns an authentication cookie
`logout`	Invalidates an existing cookie
`get`	Returns a set of tuples
`suggest`	Returns a set of atoms who field values start with a given substring
`submit`	Submits a transaction

Table 3.1: RPC Methods

To promote decoupling of the front-end and back-end, these five methods are described in detail using the exact Java syntax needed to invoke them via the GWT-RPC library. Suitable alternative libraries can be used to access the back-end from clients written in different programming languages.

The first argument `app` is common to all five method calls and indicates which website this request is intended for. When the servlet receives this method call, it queries the Servlet container to locate a ServletContext object with that application name and rejects if no such ServletContext can be found. This allows multiple websites to be reside at the same IP address.

Alloy `Int` atoms, `String` atoms, and user-defined atoms are all encoded and passed using Java String objects using the following encoding scheme:

- If the transmitted value begins with a digit or with a negation symbol, it represents an Alloy `Int` atom.

  For example, the Java String "-7" passed during an RPC call represents the Alloy `Int` atom -7.

- If the value begins and ends with double quotes, it represents an Alloy `String` atom. In the Alloy model, `String` atoms are always surrounded by double quotes. When displayed by the front-end client automatically generated by Weballoy, the double quotes are removed.

  For example, the Java String "`"Hello"`" represents the Alloy `String` atom `"Hello"`.

- If the value begins with any other character, it represents an atom corresponding to a user-defined signature, and it must start with the name of the signature, followed by a dollar symbol "`$`", followed by an integer. The integer is chosen at atom construction time to ensure the atom name does not conflict with an existing atom in that signature, but has no other significance.

  For example, the Java String "`Employee$7`" represents an atom in the `Employee` signature.

3.6.1 `login(String app, String email, String password)`

Description:

When this call is received, the servlet queries the data store to find a `LoginUser` atom whose `email` field matches the `email` argument. Since the `email` field is declared to be disjoint in the Weballoy module (Figure 2-2), there is at most one user with that email address.

As mentioned earlier, the server maintains a mapping between each valid cookie and its associated atom. If the above query returns exactly one atom, its `suspended` field is false, and its `password` field matches the input, then the server will associate that atom with a fresh cookie. On the other hand, if any of the previous condition fails, this request is rejected.

To avoid sending unencrypted passwords through the network, this method should be invoked using the HTTPS protocol rather than HTTP.

Effect:

The binary relation `String->LoginUser` maintained by the server will be overridden with a new tuple (cookie, user) if a `LoginUser` atom exists and satisfies the previously mentioned conditions. Otherwise this call has no effect.

Return values:

If the request is successful, the `LoginUser` atom will be returned to the caller, along with all its field values that are not prohibited by the access policy (as if the caller issued a `get()` RPC call with that atom). A fresh cookie returned via the "HTTP Set-Cookie" reply header is used for subsequent authentication with the server, and the `LoginUser` atom is returned solely for informational purposes. If the request is denied, an error message is returned instead.

3.6.2 `logout(String app)`

Description:

Each cookie has a server-configurable inactivity timer: if no requests are made using a given cookie for over a certain period of time, the cookie expires automatically. However, clients can explicitly request that a cookie be invalidated immediately by calling the `logout()` method. The `app` argument is the same as the other RPC calls, and there are no other arguments.

Effect:

If the cookie passed in during this call is currently associated with a valid `LoginUser`, the association is removed. If the cookie is not found or has already expired, this call has no effect.

Return values:

None.

3.6.3 get(String app, String[] atoms, String[] fields)

Description:

The atoms and fields arguments represent the *RS* set described in Section 2.2.2 and must have exactly the same number of elements. Each atom/field pair (atoms[i], fields[i]) is a query for that atom's field value in the database.

For example, the query shown in Section 2.2.2 (which asks for the courses taught by teacher Teacher$1 and the teachers teaching the course Course$1) corresponds to the following RPC call:

```
get(app, {"Teacher$1", "Course$1"}, {"Teacher$teaches", "Course$teacher"});
```

Since the get() method is called to display every dynamic value on a web page, its interface permits queries of multiple atoms and fields to be combine into a single HTTP message in order to amortize the overhead associated with the call. For example, when displaying the list of all teachers, their names can be queried in a single call.

Effect:

None.

Return values:

If at least one atom/field pair violates the access policy, an error message is returned. Otherwise, the corresponding tuples are retrieved then returned to the caller as a map from fields to tuple contents.

Example:

Consider the LoginUser signature provided in the Weballoy module (Figure 2-2). If there are at least two users LoginUser$1 and LoginUser$2, their email addresses can be queried using the following RPC call:

```
get(app,
    {"LoginUser$1", "LoginUser$2"},
    {"LoginUser$email", "LoginUser$email"}
);
```

Suppose the request is approved, and their email addresses are "a@mit.edu" and "b@mit.edu" respectively, the call will return the following map:

```
{"LoginUser$email" ->
    {
      "LoginUser$1" -> "a@mit.edu",
      "LoginUser$2" -> "b@mit.edu"
    }
}
```

3.6.4 `suggest(String app, String sig, String field, String hint)`

Description:

Modern websites using advanced JavaScript techniques often offer text boxes that automatically suggest likely values based on partial input before the user has finished entering the full value. These "suggestion boxes" use client-side JavaScript code that is triggered on key-press events, and queries the server for all possible values whose string content starts with the current text in the suggestion box.

While this functionality can be implemented with a database query retrieving all values and then filtering out the few matches on the client side, this would be very inefficient and consume unnecessary network bandwidth. Instead, these websites generally have a dedicated server script that takes the partial input ("hint") and searches the database using a custom SQL query containing that hint. This utilizes the indexing already performed by the database and reduces the amount of processing needed by the server code or by the client-side JavaScript.

To support this feature, Weballoy provides a `suggest()` method that takes two meta atoms representing a signature and a field in that signature, respectively, and the hint.

Effect:

None.

Return values:

Upon receiving the request `suggest(app, s, f, h)`, the servlet queries the database for the set of atoms in signature `s` whose value in field `f` contains at least one tuple whose first atom begins with the String `h`. The access policy checks are performed, and the result is filtered to remove any atom whose `f` value is unreadable by the user. Since the client code is likely going to display the actual values after the call, this method returns the filtered result along with their values in field `f` (as if the caller issued a `get()` RPC call with those atoms) in order to reduce one round-trip.

This method does not return an error message for access violations; even if the caller is not authorized to see any atom, this simply returns an empty map.

Example:

Consider the `LoginUser` signature provided in the `Weballoy` module (Figure 2-2). If there are three users `LoginUser$1`, `LoginUser$2`, and `LoginUser$3` whose names are "`Alice`", "`Alex`", and "`Bob`" respectively, then a suggestion box for selecting a user from the database might issue the following RPC call once the partial text "`A`" is entered:

```
suggest(app, "LoginUser", "Entity$name", "A");
```

In this case there are two candidate atoms. Assuming the caller is allowed to see both atoms and their values, this call will return the following map:

```
{"Entity$name" -> { "LoginUser$1"->"Alice", "LoginUser$2"->"Alex" }}
```

3.6.5 `submit(String app, String[][] transaction)`

Description:

The `transaction` argument represents the *AS* and *DS* sets described in Section 2.2.1 and contains an array of tuples (where each tuple is presented by an array of String).

Each tuple must match one of the following four patterns:

- (`"A"`, `"A"`, meta sig s, atom x) creates a new atom x in the signature corresponding to the meta atom s.
- (`"D"`, `"D"`, atom x) destroys an existing atom x.
- (`"A"`, atom x, meta field f, atom x_1, atom x_2, atom x_3, $\cdots$) inserts the tuple (x, x_1, x_2, x_3, $\cdots$) into the field corresponding to the meta atom f.
- (`"D"`, atom x, meta field f, atom x_1, atom x_2, atom x_3, $\cdots$) deletes the tuple (x, x_1, x_2, x_3, $\cdots$) from the field corresponding to the meta atom f.

Since the String signature has no fields, the tuple insertion/deletion pattern is never legal on a String atom; therefore, the four patterns above can be distinguished lexically just by examining the first two elements of the array, and the incoming transaction can be partitioned into the *AS* and *DS* sets without performing a database query.

Another optimization involves the allocation of fresh atoms. As stated earlier, each atom creation request must specify a fresh atom; if that atom already exists in the database, the request will be rejected as malformed. That would mean the client-code must first issue a query to find atoms that are unused, then construct the transaction using them; the client would have to repeat this process if its chosen atom got allocated by another client before this client could issue its request.

To avoid this extra round-trip query, for every atom x in an incoming (`"A"`, `"A"`, meta sig s, atom x), the servlet allocates a fresh atom y, then replaces every occurrence of x with y in the incoming transaction. This allows the client to specify an arbitrary atom to be created, as long as the chosen atom does not match one of the existing atoms referred to in this transaction.

Effect:

The incoming list of tuples is first partitioned into the two sets *AS* and *DS*, the atom creation requests are renamed as mentioned above, then the transaction is processed according to the pseudocode shown in Figure 2-3.

Return values:

If the request is approved, this call returns an empty String. Otherwise, an error message is returned which may describe which consistency constraint is violated or which insertion or deletion request is denied.

Example:

Consider the following `Employee` signature:

```
sig Employee {
   name: String,
   boss: lone Employee
}
```

Suppose the current database content is as follows:

```
Employee = { Employee$1 }
Employee$1.name = { "Alex" }
Employee$1.boss = { }
```

The following RPC call creates two new employees named Bob and Carl, with Alex as Bob's boss, and Bob as Carl's boss:

```
submit(app, {
   ("A", "A", Employee$, x),
   ("A", "A", Employee$, y),
   ("A", x, Employee$name, "Bob"),
   ("A", y, Employee$name, "Carl"),
   ("A", x, Employee$boss, Employee$1)
   ("A", y, Employee$boss, x)
}
```

Upon receiving this call, the servlet determines that there are two atom creation requests in this transaction, so the servlet generates two new unique atoms. Assuming the two new atoms are `Employee$7` and `Employee$8`, the servlet then systematically replaces all `x` with `Employee$7`, and all `y` with `Employee$8`. The new transaction is as follows:

```
submit(app, {
  ("A", "A", Employee$, Employee$7),
  ("A", "A", Employee$, Employee$8),
  ("A", Employee$7, Employee$name, "Bob"),
  ("A", Employee$8, Employee$name, "Carl"),
  ("A", Employee$7, Employee$boss, Employee$1)
  ("A", Employee$8, Employee$boss, Employee$7)
}
```

In particular, even though `Employee$8` did not exist beforehand, this transaction is able to create `Employee$8` and add it as a field value of another new atom (`Employee$7`) within a single transaction without requiring a separate round-trip communication.

3.7 Front-End

As mentioned previously, the generated server code can serve both web page requests and web service requests. The choice of front-end is independent since the server performs the access control and responds to predefined RPC messages.

By default, Weballoy automatically generates a dynamic JavaScript front-end based on the Google Web Toolkit [22] for displaying the contents of a website using the multiplicity markings and the type of each field to determine the widget used for displaying that field. Fields declared with the `lone` or `one` multiplicity markers are displayed using individual text boxes by default, whereas fields declared with `some` or `set` are displayed using a dynamic table. Users can click on a text box to edit its contents or click between cells in a table to dynamically insert or remove entries.

The auto-generated front-end and an Alloy-aware template language for customizing its appearance are described in detail in Chapter 4.

3.8 Related Work

3.8.1 Model-Driven Architecture

Model-Driven Architecture (MDA)[46] has been proposed as a cost-effective method of developing certain classes of software. The idea is to first produce the initial design in a high level modeling notation or language, then write or derive code from the model.

There are many well-known benefits for using MDA. First, it reduces the cost of evolving a software system. It is typically cheaper to modify an abstract model than the code. Second, the effort of writing a concise and precise model can often reveal bugs and inconsistencies in the design before implementation. Automation tools exist that attempt to verify high level properties on the models directly. Third, greater automation reduces the number of places where human errors could occur. This in turn reduces the cost of testing.

In practice, there are two main challenges to using MDA. First, the model often lacks a sufficient level of detail and precision. Existing specification languages are either too low-level and tedious to use, or too high-level and imprecise for fully automatic code generation. Second, the generated code tends to be unreadable and unmodifiable by humans. When the code is customized, and the abstract model changes, the user has to regenerate the code and then reapply the customization.

Many MDA tools currently offer only limited automation, such as generating skeleton code from UML models or transforming higher level Platform Independent Models (PIM) into Platform Specific Models (PSM). However, some MDA tools exist that offer limited code synthesis.

HUGO[37] translates general UML State Machines into Java by representing each state as a separate object and using a greedy algorithm to dynamically select the appropriate state transitions. Unfortunately, the UML model usually still needs to be annotated with Java code which will be automatically inserted into the guards, entry actions, or exit actions in the generated code.

Fons et al.[19] proposes an extension to object-oriented development for systematic design and construction of web applications. However, their approach provides little automation and does not ensure the access policies are enforced.

3.8.2 Website Builders

Website frameworks such as Ruby on Rails[51] and Django[9] support rapid website development by taking a high-level data model and automatically generating a skeleton website.

Ruby on Rails[51] is a Ruby framework based on the model-view-controller pattern and uses the Active Record implementation to provide transparent object-relation mapping between the Ruby code and an underlying database. By a process known as "scaffolding", Ruby on Rails can generate minimal view and controller code automatically from the model. Further customization requires direct modification of the generated Ruby code.

Django[9] is a Python framework also based on the model-view-controller pattern. Unlike Ruby on Rails, Django is much more modular and offers more flexibility at the cost of offering less automation for website designers. Whereas Rails, by design, makes many fundamental assumptions, Django emphasizes loose coupling and has alternative implementations for many core components.

Compared with Weballoy, these two frameworks are better suited for interacting with external systems, since the website designer can use Ruby or Python to access operating system calls or to execute external programs. On the other hand, for data-centric policy-rich websites where the emphasis is on accessing and modifying rich data in a secure way, the Weballoy approach is a better match. Both Ruby on Rails and Django support only simple access policies by default; custom access policies must be specified by writing additional Ruby or Python code which cannot be mechanically verified and may be subverted by other code on the website.

3.8.3 Code Generation from Alloy

In the context of Alloy, several projects aim to translate Alloy models directly into executable code. Ferreira et al.[15] translates finite state machine descriptions in Alloy into embedded executable code but does not handle arbitrary Alloy inputs. Alchemy[41] and Alloy^{II}[20] both synthesize executable code by augmenting the Alloy syntax with state mutation features or assumptions. That means the input language is less general than standard Alloy and requires all state transition operations to be encoded explicitly.

In contrast, Weballoy does not assume a specific execution paradigm since its input models describe legal states rather than legal sequences of states. Weballoy allows the use of general Alloy expressions for describing the current state and assumes any state transition is possible if it does not violate the policy.

This separation of concerns enables concise data model and policy specification for the back-end while allowing the website to be safely accessed by arbitrary untrusted client programs such as web browsers or web service portals. Instead of explicitly encoding, in Alloy, the different operations that website visitors may perform, website designers can describe them using existing web form generators[59], presentation templates, or other automated CRUD[36] generators without the danger of policy violation or the need to modify the Alloy model.

Chapter 4

Front-End Customization using Presentation Templates

The preceding chapters of this thesis focused on access policy modeling and automatic enforcement by the server; the user interface was intentionally decoupled. By providing a simple but sufficient RPC interface, the same back-end can service both web browser and web service requests. With a suitable RPC library, existing web page template tools such as XSLT[61], JSF[31], and PHPTemplate[48] can be used to construct the log in, log out, data entry, and data query web pages needed by the generated website.

However, to facilitate fully-automatic website synthesis, Weballoy can generate a default user interface, based on the Alloy data model, for navigating the website and modifying its contents. The user interface can be further customized to provide the exact look-and-feel required for a particular website. A simple template system developed for Weballoy allows website designers to embed Alloy expressions in arbitrary HTML and provides a large selection of built-in widgets such as drop-down menus, suggestion text boxes, and rich text editors.

4.1 Design Considerations

There were two design goals for the generated front-end:

Goal 1: Since the default user interface is not always ideal, the generated front-end must be customizable.

Goal 2: The customized front-end must be able to cope with changes in the underlying data model. Website requirements evolve over time, and it would be unacceptable if every change to the data model results in a freshly generated front-end without any of the prior customizations done by the website designer.

To address these concerns, I designed a simple but versatile Weballoy template language as a proof of concept where Alloy expressions may be embedded in arbitrary HTML code. The implementation thus consists of two components: (1) a heuristics-based template generator takes the data model given in Alloy and automatically generates a default template that provides the standard create, read, update, and delete operations for the website, and (2) a template rendering engine that reads the template file and displays the website contents at run-time.

The first goal is to allow arbitrary customization. By having a simple and readable template file as an intermediary between the data model and the user interface, the website can be customized by editing this template file if the default interface suggested by the heuristics is unsatisfactory.

The second goal is to allow the data model and the customization to evolve in parallel. As explained later in Section 4.2, Alloy expressions in the template file must obey certain type and scoping restrictions. As a result, the template can be statically analyzed to find all references to each signature and each field, thereby allowing several possible strategies for preserving customizations after the data model changes:

- **Automatic Refactoring:** If the Alloy data model is edited in a syntax-directed text editor, such as the Alloy4 Eclipse Plug-in, each signature or field renaming in the model can be automatically applied to the template file as well.

- **Partial Template Regeneration:** When the model is modified, the default template generator can be used to generate two fresh default template files: one based on the original model and one based on the modified model. These two templates can be textually compared to produce a file patch. This patch can then be automatically or semi-automatically applied to the actual template file that the website designer has already customized.

- **Ignore Deleted Signatures and Fields:** References to a nonexistent signature or field in the template always return the empty set. Sometimes this may be exactly the desired behavior. Otherwise, these "dangling references" can be statically determined just by parsing the template file and then flagged for manual inspection and editing by the website designer.

The Weballoy tool currently implements the last strategy, though the other two strategies may be implemented in future versions of the tool.

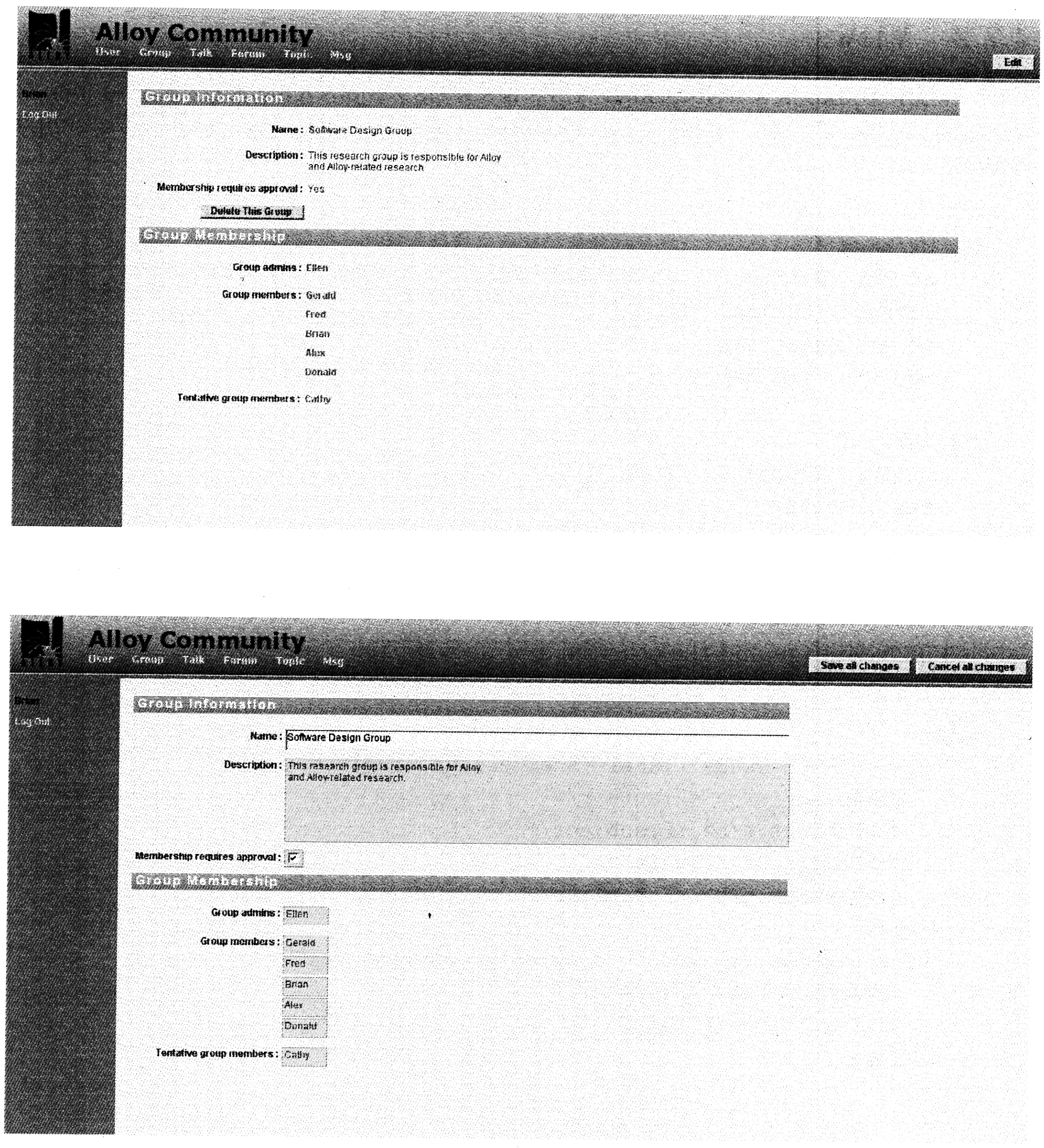
4.2 Weballoy Template Language

4.2.1 Overview

The semantics is based on defining a set of views and then displaying one view at a time. Each view is a block containing custom HTML, JavaScript, or plain text fragments which will be displayed as-is when that view is displayed.

In addition, each view can be associated with a signature and be used for displaying atoms from that signature. Given a view and a particular atom, special elements defined in the view will be substituted at run time with the current field values of that atom. Restricting the scope of each view to a single signature enables optimizations that can reduce the number of RPC calls required for displaying a given web page.

Finally, the display is always in one of two modes: viewing or editing (see Figure 4-1). In viewing mode, a user can navigate the website by starting with a given view or a given atom, then clicking on a field value that takes the user to a different atom or view.



Group Information

Name: Software Design Group

Description: This research group is responsible for Alloy and Alloy-related research.

Membership requires approval: Yes

Delete This Group

Group Membership

Group admins: Ellen

Group members: Gerald, Fred, Brian, Alex, Donald

Tentative group members: Cathy

Group Information

Name: Software Design Group

Description: This research group is responsible for Alloy and Alloy-related research.

Membership requires approval: ☑

Group Membership

Group admins: Ellen

Group members: Gerald, Fred, Brian, Alex, Donald

Tentative group members: Cathy

Figure 4-1: Viewing Mode versus Editing Mode

At any point in time, the user can click the "edit" button to enter the editing mode, where every dynamic value currently shown on the screen gets turned into text boxes or other dynamic widgets. The user can alter text box contents, add or delete rows from dynamic tables, click the "cancel" button to undo all changes made so far on this page, or click the "save" button that submits the changes to the website. The transaction, if approved, will result in a page refresh; if it violates a consistency constraint or the access policy, an error message will be displayed instead.

4.2.2 Views

Each template file is a well-formed XML file with a root element containing one or more elements of type "view". For example, consider this data model of a simple bulletin board:

```
sig Message {
    author: String,
    subject: String,
    text: LongString,
    replies: set Reply
}
sig Reply {
    author: String,
    text: String
}
```

The following example is a possible template for this model with two views, one labeled `showMessage` and the other labeled `showReply`:

```
<html>
   <view sig="Message" label="showMessage" public="yes">
      <b>Author:</b> <F:author/>
      <b>Subject:</b> <F:subject/>
      <b>Text:</b> <F:text/>
      <b>Replies:</b> <F:replies/>
   </view>
   <view sig="Reply" label="showReply" public="yes">
      <b>Author:</b> <F:author/>
      <b>Text:</b> <F:text/>
   </view>
</html>
```

The `sig="Message"` attribute indicates the field references inside the first view refer to fields in the `Message` signature. The `label` attribute is used to uniquely identify a view since multiple views may be defined for displaying atoms of the same signature.

When a view is selected for display, each regular HTML element and plain text fragment is rendered as-is. JavaScript code and CSS style sheets are permitted as long as they don't directly interfere with the rendering process, so the appearance of the generated website is fully customizable.

However, each view can also be associated with a particular Alloy signature and contain references to fields of that signature. For example, the `showMessage` view shown above is associated with the `Message` signature, and the element `<F:author/>` will be rendered as a text label (or an editable text box when the user is editing the contents) containing the `author` value of a `Message` atom.

4.2.3 Alternate Views

When an Alloy field ranges over a signature other than one of the basic types such as Int, Bool, String, or Date, each atom in its tuples will be displayed as a clickable hyperlink to that atom by default. However, it is often preferable to define a custom display format. For example, the template shown in the previous section may render a given message like this:

```
Author: John Smith
Subject: Car for Sale
Text: 2005 Ford in good condition for sale
Replies: Reply$1, Reply$3, Reply$4, Reply$5
```

Notice that each reply is rendered by giving the label of that atom rather than displayed in-line as is customary for bulletin boards. Section 4.4 describes a heuristic for choosing a better default display for atoms, but website designers can choose to define an alternate view for rendering tuples from a field.

```
<html>
   <view sig="Message" label="showMessage" public="yes">
      <b>Author:</b> <F:author/>
      <b>Subject:</b> <F:subject/>
      <b>Text:</b> <F:text/>
      <F:replies view="inlineReply"/>
   </view>
   <view sig="Reply" label="inlineReply">
      <F:author/> wrote "<F:text/>"
   </view>
</html>
```

When displaying a message, each `Reply` atom in its `replies` field will be displayed using the view labeled `inlineReply`. There is no hierarchical relationship between views, and each view can use any other view as long as there is no infinite recursion. In this case, since the `inlineReply` view is only intended for use by another view, the web designer can designate this view as non-public. That means users cannot explicitly request, from his or her web browser, that an arbitrary `Reply` atom should be displayed using that view. Given the same example, the modified template now produces the following output:

```
Author: John Smith
Subject: Car for Sale
Text: 2005 Ford in good condition for sale
Adam wrote "What is your asking price?"
John wrote "I'm currently asking for $12000"
Adam wrote "When was the last maintenance done?"
John wrote "About two months ago"
```

4.2.4 `<F:FieldName>`

By default, each field is displayed using a built-in widget designed for that field's type. For example, a String value is shown as a text label and can be edited by presenting a text box. Likewise, an Int value is shown as a text label but is edited by a validating text box which rejects the input if it is not an integer. However, these choices can be overridden if necessary. Given an Alloy field `x`, the element `<F:x/>` displaying `x` can be customized in the following ways:

- `<F:x one="yes"/>` means the text box must not be empty; if the user leaves the field blank, an error message will be shown. This attribute is implied if field `x` is already declared as `one` in the data model.
- `<F:x lone="yes"/>` means the text box can be empty. If the data model actually requires this field to be nonempty, users will not be notified of the error until they attempt to commit the changes to the database.
- `<F:x set="yes"/>` means this field will be displayed by a dynamic table of boxes. Initially the table will be populated by the actual tuples in the field. During editing, users can delete an entry by clearing its corresponding text box or by clicking the "delete" icon in front of that row in the table. Clicking between two rows or at the top or bottom border of the table will insert a new empty text box at that location.
- `<F:x some="yes"/>` is the same as `<F:x set="yes"/>` except the user will be shown an error message if this field does not contain at least one entry.
- `<F:x ro="yes"/>` means this field is read-only in this particular view and cannot be edited. This attribute is automatically implied if the field is an Alloy *defined* field whose value is automatically derived from other fields.
- `<F:x view="v"/>` means this field should use the built-in widget named "v" (if v is String or Int), or a user-defined view labeled "v".
- `<F:x view="v" sort="+name,-salary"/>` is the same as above, except the tuples will be sorted before being presented. In this case, the range of the relation (which must be binary) is first sorted in ascending order by their names. Entries with the same name will be ordered in descending order by their salary. If both fields are equal, they will be presented in their current order as stored in the database. As a proof-of-concept, this sorting syntax is very basic and does not support higher arity relations or user-defined comparator. A future version of the Weballoy template language may support more advanced features.

These attributes may be combined. For example, field `x` can be displayed as a set, and be read-only at the same time by using the element `<F:x set="yes" ro="yes"/>`.

4.2.5 `<F:S:SigName>`

Similarly, the element `<F:S:signame view="viewname"/>` displays each atom of the signature named "`signame`" using the view labeled "`viewname`".

4.2.6 Widgets

The current implementation provides the following widget types. The template can insist that a value be displayed using a particular widget by using the `view="..."` attribute described earlier.

- String: When displayed, this uses an in-line `SPAN` element with CSS style `a4textbox`; furthermore, when the box is empty, the element will be tagged with an additional CSS style `a4textboxEmpty` allowing designers to provide a special appearance for empty values. When this widget is being edited, it uses a single-line `INPUT` element with the same CSS style.

- LongString: When displayed, this widget behaves in the same way as String. When edited, it uses a multi-line `TEXTAREA` element with CSS style `a4textbox`.

- RichString: When displayed, it behaves in the same way as String. When edited, it uses the rich text editor provided by the Google Web Toolkit, which supports arbitrary colors, font styles, and font sizes.

- Int: When displayed, this widget converts each tuple into an integer and displays it; if the value in the database is nonempty but not a valid integer, it is currently displayed as 0. When edited, it uses a single-line text box which rejects the input if it is not a well-formed integer. The box can be left empty if either `lone="yes"` or `set="yes"` is specified.

- Date: When displayed, this widget regards each value as a number representing the number of seconds since January 1 of 1970. The JavaScript code running in the user's browser converts this number into a year/month/day representation using the user's time zone and locale preferences. When edited, it uses a single-line text box which rejects the input if it is not well-formed. If the user specified hours, minutes, and seconds information in the text box, they are ignored when this value is saved.

- DateTime: This behaves just like the Date widget, except it will display and accept the hours, minutes, and seconds information.

- Bool: When displayed, this widget uses an in-line `SPAN` element with the CSS style `a4bool`. If the value from the database contains both True and False atoms, and possibly other atoms, the widget will display "Both". If the value contains at most one Bool atom, it is shown as either "Yes" or "No" correspondingly. Otherwise, it is shown as "`--`" by default, meaning the field does not contain any Bool atom. When edited, it uses a check box by default if the field's multiplicity is declared to be "one". Otherwise, it uses a combo box showing up to four choices: "`--`", "Yes", "No", and "Both" (the empty choice is shown if the field is `lone` or `set`, and the "Both" choice is shown if the field is `some` or `set`).

These widgets are defined with explicit CSS style names to make it easier for customization. If a widget needs to be displayed in two different ways in the same website, it can be done by wrapping the `<F:x/>` element inside a `<SPAN class="someClass">` element, then the CSS style sheet can differentiate between the elements nested in a `SPAN` element with one style name versus those nested in an element with a different style name.

4.2.7 URL Anchor

Since the template can contain multiple views, the renderer uses the fragment URL [4] to decide at run time which view and which atom to display. The portion of the URL before the "#" symbol determines the web server's IP address and the application identifier (described in Section ??). The portion after the "#" symbol determines the view and an atom.

For example, the URL `http://alloy.mit.edu/community#showUsers` instructs the application on `alloy.mit.edu` with the identifier "`community`" to display the view labeled "`showUsers`".

Likewise, the URL `http://alloy.mit.edu/community#showUser.User$8` will display the atom `User$8` using the view labeled "`showUser`". The table below lists the different formats currently supported:

Pattern	Effect
	Display the default view (the view with no or empty label)
`#view`	Display the view labeled "`view`" in viewing mode
`#view+`	Display the view labeled "`view`" in editing mode
`#view.atom`	Show the atom "`atom`" in viewing mode using the view "`view`"
`#view.atom+`	Show the atom "`atom`" in editing mode using the view "`view`"
`#view+sig+`	Create a new atom for signature "`sig`" using the view "`view`"

Table 4.1: URL Anchor Patterns

4.2.8 Editing Modes and Action Buttons

As mentioned earlier, at any given point in time, each page is always in viewing mode or editing mode (see Figure 4-1). To enable users to edit the contents on a page, suitable hyperlinks can be placed into a view where the target of the link matches one of the patterns (Table 4.1) for entering viewing mode or editing mode. Website designers can customize the appearance of these hyperlinks using CSS.

Alternatively, Weballoy template language also supports buttons that perform different functions based on its "`magic`" attribute. For example, the button `<input type="button" magic="edit"/>` takes the user from the viewing mode into the editing mode, and the button `<input type="button" magic="save"/>` attempts to save the changes on that page and go back to the viewing mode.

The types of action buttons currently supported are as follows.

- `<input type="button" magic="edit"/>`

  This causes the current page to enter the editing mode.

- `<input type="button" magic="save"/>`

  This saves the changes made on this page so far.

- `<input type="button" magic="cancel"/>`

  This cancels the changes made on this page since entering the editing mode, then returns the user back to the viewing mode.

- `<input type="button" magic="back"/>`

  This is equivalent to pressing the "back" button on most browsers and will take the user to the previous page.

- `<input type="button" magic="delete"/>`

  This prompts the user for confirmation before asking the server to delete the atom currently being displayed.

- `<input type="button" magic="go" view="anchor"/>`

  This prompts the user to save the changes made on this page so far, if any, then navigates the browser to display the given anchor (where the anchor must match one of the patterns in table 4.1). In addition, each occurrence of the "`*`" character in the "anchor" attribute will be replaced by the current atom being displayed. For example, if the current atom is `User$3`, pressing the action button `<input type="button" magic="go" view="showUser.*">` will navigate to the URL `showUser.User$3` thereby displaying the atom `User$3` using the view labeled `showUser`.

4.2.9 Mode Filtering using CSS

By default, when displaying a web page, the very same layout is used for both the viewing mode and editing mode; entering the editing mode involves simply dynamically replacing each Alloy value displayed on screen with a suitable text box or other widget. Alternatively, the two modes can be displayed very differently by defining two distinct views, one for editing and one for viewing.

However, sometimes the two views are very similar and it would be ideal to share the same view without needless duplication. To enable this, the template renderer places the entire web page inside a `<SPAN class="a4editing"/>` element when in editing mode, and inside `<SPAN class="a4viewing"/>` when in viewing mode. Therefore, elements can be made invisible or be shown differently by using CSS to differentiate between the two modes.

4.3 Implementation of the Rendering Engine

The current implementation for rendering the template at run time to produce each web page is a single Google Web Toolkit [22] module.

A GWT module usually corresponds to a fragment URL [4] whose dynamic behavior depends on the anchor identifier after the "#" symbol in the URL. For example, the URL `http://www.google.com/showHelp#language` might instruct GWT code at that website to display help messages regarding different language settings, whereas a different URL `http://www.google.com/showHelp#calendar` might display tips about the calendar settings instead.

In the case of the Weballoy template rendering engine, the anchor identifier is used to determine which view and which atom to show (table 4.1). When the user first visits an Weballoy-generated web page, the entire JavaScript file generated by GWT is loaded into the web browser. The code determines the chosen view then renders it by structural recursion: each HTML and plain text fragment is rendered as-is, each Weballoy specific element is displayed with respect to the current atom being displayed, and subviews are rendered by invoking the renderer recursively.

4.3.1 Queries

Alloy expressions in the template file are resolved by issuing one or more `get()` requests to the server. The structured nature of the template language, given the fact that each view can only refer to fields of exactly one atom, makes it often possible to combine multiple `get()` requests and reduce the number of round-trip communication with the server.

For example, to display every field of atom `x`, first it is necessary to get all its field values. Fields do not depend on one another, so they can be combined into one request. Suppose one of its fields contains two atoms `a1` and `a2`, and that field is to be rendered using a view labeled `v`. Since rendering `v` on `a1` cannot possibly depend on any value retrieved for rendering `v` on `a2`, the renderer can generate a single request for both `a1` and `a2`'s field values.

4.3.2 Widgets

Each widget stores its originating atom and field as well as its initial and current value. For example, suppose the renderer is displaying the atom `Paper$1`, and its `author` field contains three String values `Alex`, `Bob`, and `Carl`. At run time, three JavaScript objects `x1`, `x2`, and `x3` will be created, each remembering it came from `Paper$1.author`.

When in viewing mode, each object is displayed using a text label or other suitable static widgets. For example, `x1` displays a text label containing the text `Alex`.

When entering the editing mode, these objects hide the static widgets currently being shown and instead display editable text boxes or other dynamic widgets. For example, `x1` will display the text `Alex` using a single-line text box.

If a field has the multiplicity of `some` or `set`, it can have multiple values at the same time. Each value is displayed as a separate text box, and the user can click on the space between them to insert blank text boxes. This is done by cloning one of the existing widgets then setting its contents to blank. For this purpose, even if a field is currently empty, one text box is still generated to represent that field then made invisible.

4.3.3 Transactions

When the user clicks the "save" button, each widget is queried for its original value and the current value. The differences are transmitted to the server as a transaction.

For example, continuing from the example in the last section, suppose the user deletes the first text box `x1`, changes the second text box `x2` to contain the value `William` instead of `Bob`, and adds a new text box `x4` containing the value `David`. The partial database state known to the user now is that `Paper$1.author` has at least three Strings: `William`, `Carl`, and `David`. In the pre-state before editing began, `Paper$1.author` contains `Alex`, `Bob`, and `Carl`, so the JavaScript code will request the deletion of `Alex` and `Bob`, and the insertion of `William` and `David`.

There is a race condition here where the user's pre-state may no longer be the latest database state. Since web page visits are not transactions, and web page visits by users are often transient and intermittent, it is usually undesirable to lock the data while a user edits it. That means between the time a page is displayed for editing and the time the user submits the changes, the underlying data may have been modified by another user.

In the last example, suppose yet another user added Moriarty to the list of authors after this user loaded the screen but before this user clicks the "save" button. This user assumes that, after the save, the list of authors consists of only William, Carl, and David. But instead, his transaction will not remove Moriarty (since Moriarty is not in the pre-state that he saw), and the final database state actually contains four names: William, Carl, David, and Moriarty.

This challenge is common among all web frameworks, and there are a variety of standard approaches for addressing this. One approach is to store a serial number for each page: modifying a page increments its counter by one, and when a user attempts to save a page whose counter has advanced beyond the user's expectation then the user will be notified and be given a chance to either overwrite the intermediate change or to start over. Another approach is for the latest save to always overwrite any intermediate changes. Another possibility for some types of application is to intelligently merge the changes and only issue a warning if there is an irreconcilable merge conflict.

The current Weballoy implementation of the client front-end will merge the changes between multiple concurrent edits since each user's web browser transmits only the edits made by the user on that page; this decision was made in order to simplify the implementation effort for this proof-of-concept template language. Future versions can easily adopt the per-page change counter or other standard solutions for addressing this race condition.

4.4 Default Template Generation

The preceding sections describe the Weballoy template language and the renderer for displaying the template. Its support for arbitrary HTML, its ability to refer to Alloy fields, and its systematic use of CSS styles makes the front-end very customizable. However, it can be a daunting task to create such a template file from scratch. This section describes a very simple but effective approach of taking the data model given in Alloy and generating a initial template file for creating, reading, updating, and deleting atoms and tuples for that model (often referred to as CRUD[36] interface construction).

4.4.1 View for Creating an Atom

For each user-defined signature, the generator produces several views. The first view is a view for creating atoms of that signature. To avoid name collisions, the generator prepends the character "2" to the signature name to form the label for this view, since signature names are not allowed to start with a digit. Then it generates a table of rows where each row contains one of the signature's fields followed by a suitable widget for editing that value. For example, consider the following signature:

```
sig Employee {
    name: String,
    homeAddress: String,
    salary: Int
}
```

The template generator will generate this view:

```
<view sig="Employee" label="2Employee" public="yes">
   Add Employee
   <table>
      <tr> <td> Name:         </td> <td> <F:name/>        </td> </tr>
      <tr> <td> Home Address: </td> <td> <F:homeAddress/> </td> </tr>
      <tr> <td> Salary:       </td> <td> <F:salary/>      </td> </tr>
   </table>
   <input type="button" magic="save" value="Create the Employee"/>
   <input type="button" magic="back" value="Cancel"/>
</view>
```

Field names and signature names are automatically capitalized in the view; for example, the heading for field `name` is "Name", and the heading for field `homeAddress` is "Home Address".

Two buttons are also provided for saving this new atom (thereby creating it) and for canceling this request (by taking the user back to the page that leads them here).

4.4.2 View for Displaying Every Atom of a Signature

For each user-defined signature, the generator also produces a view for displaying every atom from that signature. This view lists the names of every field in this signature then invokes another view for display every atom. To avoid name collisions, the labels for the two views are formed by prepending "0" and "1" to the signature name, respectively. For example, given the Employee signature shown earlier, the generator produces two views for listing this signature similar to the following:

```
<view label="0Employee" public="yes">
   Employees
   <table>
      <tr>
         <th>Name</th>
         <th>Home Address</th>
         <th>Salary</th>
      </tr>
      <F:S:Employee view="1Employee"/>
   </table>
   <input type="button" magic="go" view="2Employee+Employee+"
      value="Add Employee" />
</view>

<view sig="Employee" label="1Employee">
   <tr>
      <td> <F:name/>        </td>
      <td> <F:homeAddress/> </td>
      <td> <F:salary/>      </td>
   </tr>
</view>
```

The signature and field names are converted as before; furthermore, the name of the signature is automatically pluralized as the caption for the table. After creating a TABLE element and producing the list of field names as its first row, each atom from the Employee signature is displayed using the view labeled "1Employee". Also, since the generator knows the view for creating an Employee is labeled "2Employee", a button is created for the URL "2Employee+Employee+" (meaning to show an atom creation dialog for signature Employee using the view labeled 2Employee).

4.4.3 View for Displaying One Atom

By default, for every user-defined signature, the generator produces a view (whose label is the same as the signature name) for viewing an individual atom from that signature. This view consists of a table where every row contains the label of a field followed by the field content itself. For example, the generated default view for displaying an individual atom from the Employee signature shown earlier is similar to the following:

```
<view sig="Employee" label="Employee" public="yes">
   Employee
   <table>
      <tr>
         <td> Name:     </td>
         <td> <F:name/> </td>
      </tr>
      <tr>
         <td> Home Address:    </td>
         <td> <F:homeAddress/> </td>
      </tr>
      <tr>
         <td> Salary:     </td>
         <td> <F:salary/> </td>
      </tr>
   </table>
   <input type="button" magic="delete" value="Delete This User"/>
</view>
```

This completes the description of the default template generator. It simply iterates over every user-defined signature and produce three public views and one private view for each signature. Web designers can reorder or hide certain fields, rename the labels, alter the presentation style sheets, or make more elaborate customizations as needed.

4.5 Related Work

4.5.1 Template Languages

The growing popularity of the model-view-controller pattern for website construction has resulted in the development of many different template languages and engines. Proper use of templates provides a separation of concerns and allows the same data to be presented in different formats for different users or for different web browsers.

Enterprise applications based on Java EE [27] benefit from template frameworks such as JavaServer Faces [31] which is heavily object-oriented and interoperates well with both JavaServer Pages [32] code and JavaBeans. Domain specific systems often come with suitable template languages as well; the Drupal [11] open source content management system comes with a PHP-based template engine named PHPTemplate[48] which is also usable outside of Drupal. More generally, applications with XML input or output can convert the XML data into human-readable text or HTML by writing the transformation rule in the XSLT [61] language, or by writing XQuery[60] code to extract data from XML for presentation.

The main difference with Weballoy template language is that these systems are much more powerful in that any line in the template can refer to any part of the input data; JavaServer Faces and PHPTemplate even allow arbitrary code in the template. While they offer the user more capabilities for generating the presentation, the unstructured access to data makes it impossible to statically analyze then optimize the template to minimize the number of database accesses, or to perform mechanical refactoring on the template files.

4.5.2 Ajax Libraries

The current implementation of the Weballoy template renderer uses an increasingly common technique known as Ajax (Asynchronous JavaScript and XML) for querying and submitting data to a web server in the background without requiring the disruptive and time consuming reloading of the current web page.

Initially a niche technique rare among mainstream websites, it was popularized partly by the success of Google Gmail which presents an online email client with responsiveness and user interface similar to a local email application, unlike other contemporary email services such as Microsoft Hotmail and Yahoo! Mail.

Ajax libraries such as jQuery [30], Google Web Toolkit [22], and Flapjax [17] can be used to manually build a dynamic interface for accessing remote data. For read-only access to specific data formats such as relational data presented in JSON (JavaScript Object Notation), frameworks such as Exhibit [12] can partly automate the construction of a dynamic viewing interface.

Some of these tools support automated interface construction. But unlike Weballoy, they are not integrated or driven by a formal modeling language which limits the amount of safe automation or static analysis that can be performed.

4.5.3 Interface Languages/Builders

Many languages and builders exist for describing and generating user interfaces. In the context of website construction, systems such as Django[9] and phpMyAdmin[47] offer standard administrative interfaces for reading and modifying data stored behind the website. However, their built-in user interfaces are not very customizable, unlike the Weballoy template language. Further customization of Django requires Python programming and suffers the same limitation as the other template languages mentioned earlier in this section.

DabbleDB[6] allows users to customize the display of data on-the-fly using the very same graphical user interface presented on screen. For the scenarios envisioned by the DabbleDB developers, the system allows non-programmers to conveniently modify the interface without editing any theme files or writing any line of code. However, it does not permit arbitrary customization. Furthermore, it does not support rich structural data models which further limits the types of presentation possible.

XForms[59] has been proposed as a standard for defining interactive web forms for data entry on a website. Its data model is based on XML and is designed from the start to support the model-view-controller pattern. Similar to Weballoy template language, it offers a more structured and formal approach to data access and data submission. However, its strict adherence to a data model based in XML both enhances and limits its capability at the same time: the rich document validation capability of XML allows much more powerful client-side input validation and reduces the amount of required round-trip communication to the server, but the rigid XML form display and submission semantics limit its general applicability.

Chapter 5

Case Studies

5.1 Alloy Community Website

5.1.1 Original Website

My research group maintains a website for users of the Alloy language and tools. I was involved in a redesign of that website in early 2008. One of our main objectives was to allow users to upload Alloy-related publications to the website and to foster collaborations among researchers.

To minimize our administrative efforts and to empower the users, we wanted to allow users to upload and modify contents on the website directly. But at the same time, we had to ensure users could not modify each other's contents. The access policy rules got more tricky when we introduced the notion of *research groups* where users could form and join research groups, and contents could be tagged with the groups they belong to.

The open source content management system Drupal [11] was chosen because several students in our research group had experience with it. The initial prototype was built in a week by one graduate student and two high school students who were visiting our research group at the time. Another graduate student spent one more week customizing it in response to user feedback.

However, when we wanted to introduce the notion of group tagging, we realized that our notion of group permission and ownership was very different from Drupal's, and we were unable to find any third-part Drupal add-on modules that supplied the behavior we wanted.

For example, we wanted users to be able to tag their papers with the research group they belonged to. One module we found allowed a user to tag papers with any group, even groups they did not belong to. Another module did not allow group owners to generate their own labels and required the website administrators to manually create a new label for each group.

To finish the website, we enlisted the help of a professional Drupal developer who spent 40 work hours selecting appropriate third-party Drupal add-on modules, installing and configuring them to satisfy our requirements as much as possible, then manually modifying Drupal's underlying PHP code until it fulfilled our requirements exactly.

The resulting website has been a great success for our users, but it is unsatisfactory for two reasons. First of all, its access policy is very rigid and will require further direct PHP modifications if we want to change the access policy beyond the standard paradigm. Secondly, it imposes long term maintenance costs: if we wish to upgrade to the next version of Drupal, we have to manually apply the PHP code changes again. The changes may be nontrivial if the Drupal API changes or if new Drupal components interact with our code modifications in unexpected ways.

5.1.2 Weballoy Implementation

For comparison, we modeled the core functionality and access policy of the website using Weballoy: the data model consists of 32 lines (6 signatures and 21 fields in Figure 5-2) and the access policy consists of 22 lines (11 capability clauses and 10 conditions in Figure 5-3 and 5-4).

Most of the definitions are straightforward; however, the group membership management is nontrivial. In this website, group owners can designate whether a group is open or not. Users can directly join an open group, but membership in a closed group requires confirmation by an owner of that group. Users wishing to join a closed group can add themselves to the group's tentative membership list. Super users and the group's owners can deny the requests by removing users from the tentative member list, or approve the requests by moving users from the tentative member list into the regular member list. The insertion and deletion from the three lists (owners, regulars, and tentatives) are described by six pre-conditions shown in Figure 5-4. For example, the pre-condition for insertion into the regular member list is as follows:

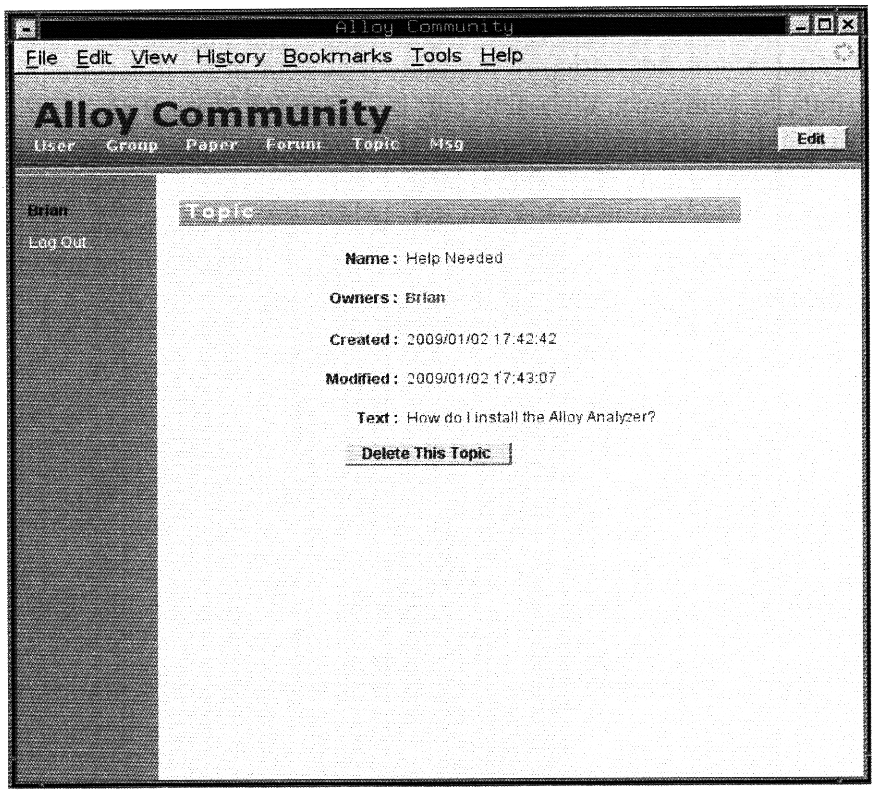
```
pred regulars.preAdd {
  let g=this.first, u=this.second |
    ((g.closed=False && me=u) || me in g.owners || me.super=True) }
}
```

This predicate is evaluated for each tuple being inserted into a group's regular member list. Each tuple is a pair of `Group` and `User` atoms, so the `let` statement is used to extract the group `g` and user `u` from the pair. This predicate returns false and thus rejects the insertion if it does not match at least one of three cases: (1) a user is allowed to insert himself if the group is not a closed group, (2) a user is allowed to insert anybody if he is an owner of the group, and (3) super users on this website are allowed to insert anybody, assuming it does not violate the capability rules or a consistency constraint.

5.1.3 Limitations

The automatically generated website currently cannot send notification emails since the current Weballoy syntax does not contain primitives for describing email transmission or retrieval. To improve responsiveness, commercial website frameworks usually do not send emails during the processing of individual web page requests; instead, outgoing emails are added to a queue to be sent at a later time by a dedicated server program. In principle, Weballoy can be extended using the same approach: a special relation can be defined to represent the queue, and user requests can generate outgoing emails by inserting into this relation.

Since the widget for handling file uploads is not yet implemented, the model does not allow users to upload files either. The generated website is otherwise complete and nearly identical in appearance after customization.


Alloy Community
File Edit View History Bookmarks Tools Help
Alloy Community
User Group Paper Forum Topic Msg
Edit
Brian
Log Out
Topic

Name : Help Needed

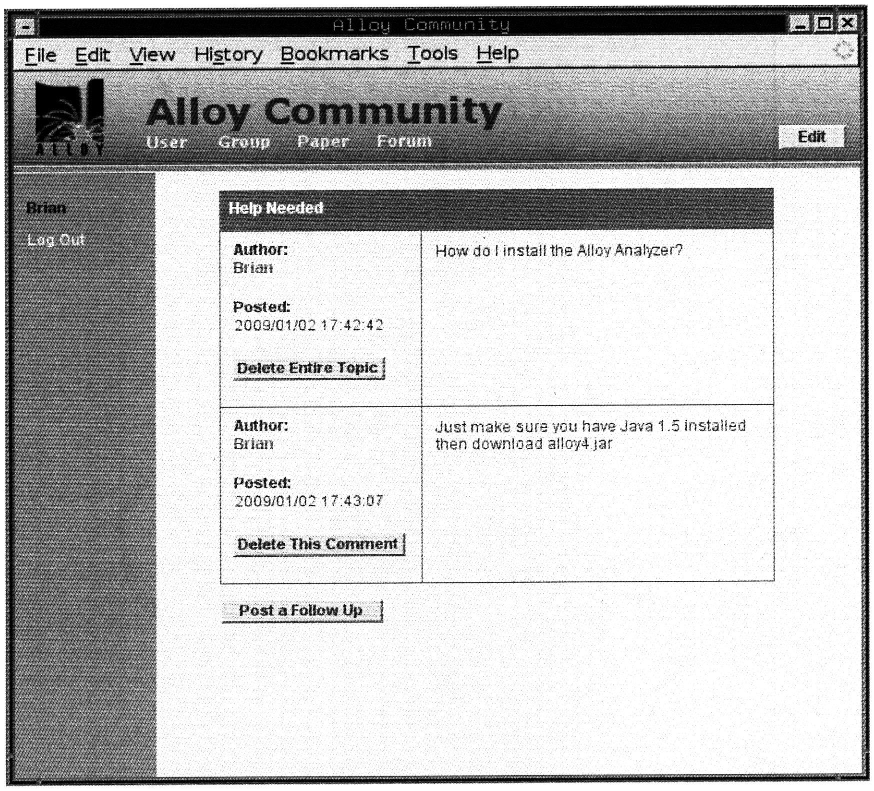
Owners : Brian

Created : 2009/01/02 17:42:42

Modified : 2009/01/02 17:43:07

Text : How do I install the Alloy Analyzer?

Before Customization



Author: Brian

Posted: 2009/01/02 17:42:42

How do I install the Alloy Analyzer?

Author: Brian

Posted: 2009/01/02 17:43:07

Just make sure you have Java 1.5 installed then download alloy4.jar

After Customization

Figure 5-1: Alloy Community Website

```
sig User extends LoginUser {
   bio     : lone LongString,
   super   : Bool,
   getMail : Bool,
   admins  = (@owners.this) :> Group,
   groups  = regulars.this
}

sig Group extends NamedEntity {
   description          : lone LongString,
   closed               : Bool,
   regulars, tentatives : set User
} {
   closed=False => no tentatives       // Group is open, so users must be either owners or regulars
   disj[owners, regulars, tentatives]  // The same user cannot have two different membership types
}

sig Paper extends NamedEntity {
   date    : Date,
   venue   : lone LongString,
   descr   : lone LongString,
   authors : some String,
   tags    : set Group
}

sig Forum extends NamedEntity {
  descr    : lone LongString,
  weight   : Int,
  readOnly : Bool,
  topics   : set Topic
}

sig Topic extends NamedEntity {
  text     : LongString,
  pinned   : Bool,
  readOnly : Bool,
  replies  : set Msg
}

sig Msg extends Entity {
  text : LongString
}

fact "Every user must have a distinct name." { all disj a, b: User | a.name != b.name }

fact "Every group must have a distinct name." { all disj a, b: Group | a.name != b.name }
```

Figure 5-2: Data model for the case study

```
// This defines the set of capabilities possessed by the user "me"
fun policy : Action->univ->univ
{
 // every user can read his/her own fields
 R -> me -> field$


 // super users can read everything.
 // Others cannot read the "email", "getMail", and "tentative group membership" fields.
 + R -> univ -> (me.super!=True => (field$ - LoginUser$email - User$getMail - Group$tentatives) else field$)


 // if you own a group, or if you're a tentative group member of a group,
 // you can see its tentative group membership field
 + R -> (owners.me + tentatives.me) -> Group$tentatives


 // every user can modify his/her email, password, and all User fields except the "super user" flag
 + W -> me -> (Entity$name + LoginUser$email + LoginUser$password + User$.fields - User$super)


 // paper owner can modify everything; topic or message owner can modify the name and text of the topic or message
 + W -> (owners.me) -> (Paper$.fields + Entity$name + Topic$text + Msg$text)


 // group owner can modify everything, including the owner field
 + W -> (me.admins) -> (Group$.fields + Entity$owners)


 // super users can modify everything; others can add themselves to group membership or tentative membership list
 + W -> univ -> (me.super=False => (Group$regulars + Group$tentatives) else me.super=True => field$ else none)


 // everybody can post new forum message or forum replies
 + A -> univ -> (Forum$topics + Topic$replies)


 // super users can create anything; non-super users cannot create new User or Forum
 + ADD -> (me.super=False => (Group$ + Paper$ + Topic$ + Msg$) else none)
 + ADD -> (me.super=True => (Group$ + Paper$ + Forum$ + Topic$ + Msg$) else none)


 // super users can delete anything.
 // non-super users can delete self, any group he/she manages, and any paper he/she owns.
 + DELETE -> (me.super!=True => (Paper&(owners.me) + Group&(owners.me)) else univ)
}
```

Figure 5-3: Capability rules for the case study

```
// only super users can create a pinned topic, or create a read-only topic, or post in a read-only forum
pred Topic.postAdd { me.super = True || (this.pinned + this.readOnly + topics.this.readOnly) = False }


// only super users can post follow-up messages to a 'read-only' topic
pred Msg.postAdd { me.super = True || replies.this.readOnly = False }


// You cannot create a new paper tagged with a group you don't belong to
pred Paper.postAdd { this.tags in (this.owners.groups + this.owners.admins) }


// You cannot change your paper's tag to a group you don't belong to
pred tags.postAdd { this.second in (this.first.owners.groups + this.first.owners.admins) }


// Only super users and a group's owners can change the owner list of a group
// Furthermore, anyone can remove him or herself from a group's owner list.
pred owners.preAdd { let g=this.first                | g in Group =>        (me in g.owners || me.super=True) }
pred owners.preDel { let g=this.first, u=this.second | g in Group => (me=u || me in g.owners || me.super=True) }


// Super users and group owners can modify a group's regular member list.
// Furthermore, anyone can join an open group or leave a group.
pred regulars.preAdd { let g=this.first, u=this.second | ((g.closed=False && me=u) || me in g.owners  || me.super=True) }
pred regulars.preDel { let g=this.first, u=this.second |                          (me=u || me in g.owners  || me.super=True) }


// Super users and group owners can modify a group's tentative member list.
// Furthermore, anyone can add or remove him or herself from a group's tentative member list.
pred tentatives.preAdd { let g=this.first, u=this.second | (me=u || me in g.owners  || me.super=True) }
pred tentatives.preDel { let g=this.first, u=this.second | (me=u || me in g.owners  || me.super=True) }
```

Figure 5-4: Pre- and post-conditions for the case study

5.2 MIT FamilyNet

5.2.1 Original Website

MIT FamilyNet[13] is an online community for families of MIT students, faculty, and staff. The vision for a family-focused networking tool came about in 2006 when the MIT Graduate Student Council housing community affairs committee and other groups realized that there was a pressing need for a centralized meeting place to address the non-academic needs of MIT families. Some examples of those needs are:

- New parents needing support from each other and in forming play groups for their children.
- International spouses and partners wanting to experience more of American culture with others.
- Newcomers to MIT wanting to know where to find goods and services.

A small task group was formed in October 2007 to come up with the requirements for MIT FamilyNet. A very thorough requirements gathering phase was conducted, resulting in a requirement document roughly 100 pages in length.

Given the limited budget, various open source content management systems were evaluated, and Drupal[11] was chosen for its relative flexibility, solid reputation, and robust developer community. For the first-pass prototype, installing and configuring various Drupal modules met roughly 60% of the requirements, with another 10% achieved by manual editing of the underlying PHP source code.

A representative cross-section of test users where invited to try out the prototype and suggest improvements. In implementing each of those changes, the FamilyNet website developer first tries to configure and install pre-written Drupal modules to meet the requirements. On many occasions, however, minor custom coding was required.

The complexity of the project was becoming apparent. The major challenges identified were as follows:

- Though the learning curve for Drupal was low from a user's standpoint, it was considerably steeper for the volunteer developer chosen to implement MIT FamilyNet.

- Many Drupal modules were poorly documented, sometimes with less than one page of documentation, making it difficult to gauge the actual functionality of a module without installing it first. Several add-on modules were misadvertized and did not fulfill the promises of their documentation, perhaps due to the difficulty of keeping the documentation current in a constantly evolving open source project.

- The quality of add-on modules varied considerably. Some of the chosen modules were only weakly supported by their current maintainers.

- Drupal modules utilized event hooks to provide an aspect-oriented architecture and encourage component reuse. However, sometimes modules interacted adversely with each other in unexpected ways, requiring the developer to manually debug and trace the underlying PHP code.

- Most significantly, the base system and the installed modules had predefined permissions that could be enabled and disabled for different user roles but could not be further customized without custom coding. The MIT committee responsible for creating the requirements document carefully produced rules that promoted open community gathering but protected the privacy of the users. Many of these requirements were impossible to implement without custom coding.

As a result of these difficulties, by the time FamilyNet concluded the final user testing, the deployed Drupal version was already one major release and three minor releases behind. When asked to summarize the experience of building the website using Drupal, the project's technical lead remarked:

> "The Drupal community pride themselves on providing plug-and-play modules that can be configured in a variety of ways. In the FamilyNet project, Drupal typically covered 70% to 90% of a particular requirement, and when the project team decided that the gap must be bridged, customization was usually the only solution. For many non-core requirements, we ended up being constrained by the Drupal modules that could meet the needs most of the way. This brings to mind the way some companies have enterprise software (such as SAP) define the business process rather than the other way around."

One example of a seemingly intuitive requirement involved the group-specific forum for each user group. Users on this website could form user groups, and only group members could post messages on the group forum. However, in order to encourage group membership and participation, it was decided that non group members should be able to view the forum messages. The former requirement of a discussion forum associated with a group was taken care of by a popular Drupal module, but the latter requirement required custom coding.

5.2.2 Weballoy Implementation

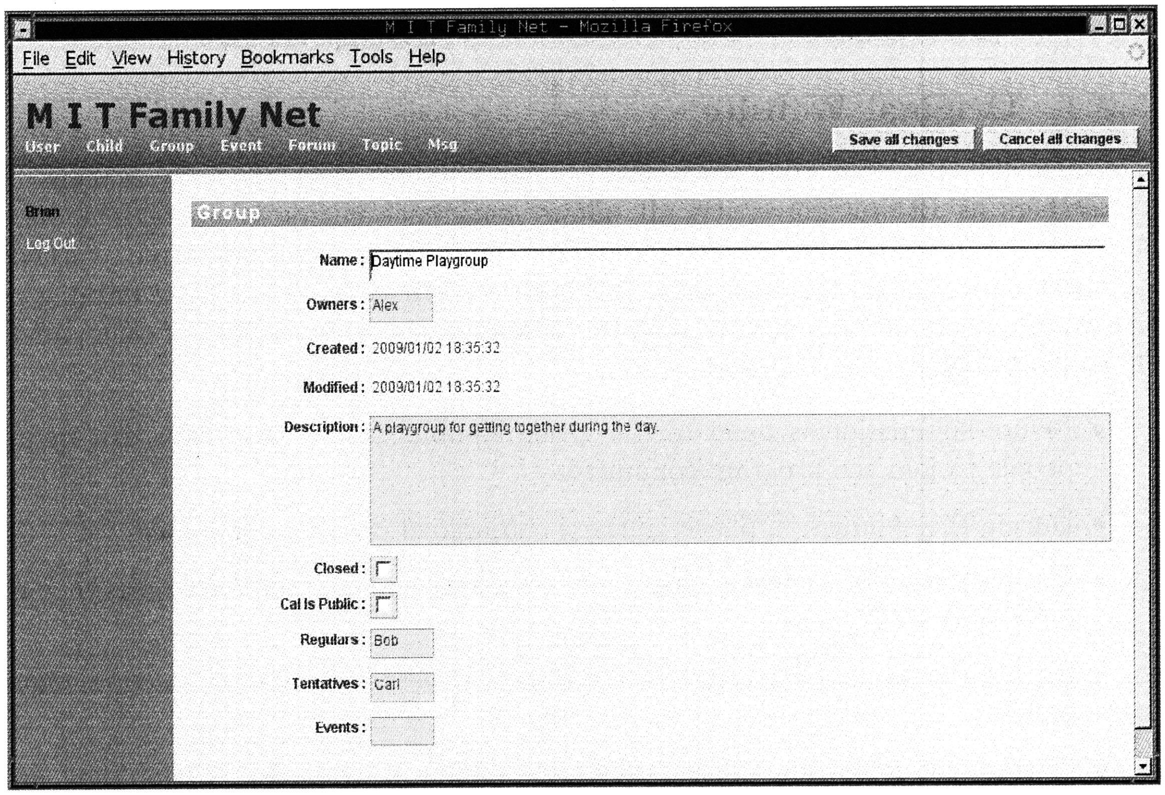
For comparison, the data model and access policies were modeled using Weballoy. The data model is large since the user profile contains many fields such as "neighborhood", "residence", "mainLanguage", "secondLanguage", and "childcareInterests".

As in the previous case study, each user group on this website could also be designated as either restricted (requiring a group owner's explicit approval) or open. This was modeled using three fields (`owners`, `regulars`, and `tentatives`) and six pre-conditions for inserting and deleting from these fields, just like the model written for the Alloy Community Website. Similarly, the `Forum`, `Topic`, and `Msg` signatures and fields were reused as-is.

The interesting case of the group forum was actually very straightforward to specify. The group module chosen for the Drupal project insisted that group-associated contents must be either readable and writable by group members only, or readable and writable by everyone. In the Weballoy model, it was easy to specify the two permissions separately.

5.2.3 Limitations

As mentioned in the previous case study, file uploads and email transmissions are yet implemented in Weballoy; therefore, the file sharing and email notification features of FamilyNet are missing from the automatically generated website. Since users cannot upload images, user icons are not supported either. The website is otherwise complete.


M I T Family Net - Mozilla Firefox
File Edit View History Bookmarks Tools Help
M I T Family Net
User Child Group Event Forum Topic Msg
Save all changes
Cancel all changes
Brian
Log Out
Group

Name : Daytime Playgroup

Owners : Alex

Created : 2009/01/02 18:35:32

Modified : 2009/01/02 18:35:32

Description : A playgroup for getting together during the day.

Closed :

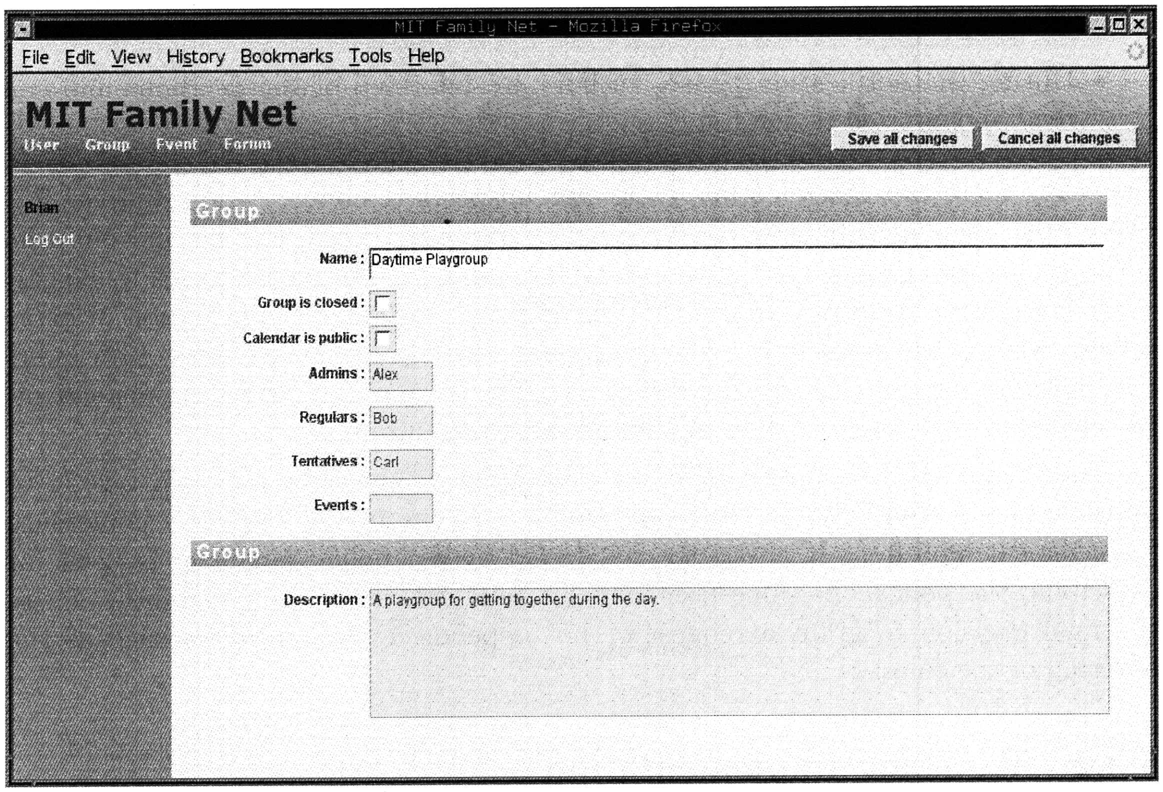
Cal Is Public :

Regulars : Bob

Tentatives : Carl

Events :

Before Customization



Name : Daytime Playgroup

Group is closed :

Calendar is public :

Admins : Alex

Regulars : Bob

Tentatives : Carl

Events :

Description : A playgroup for getting together during the day.

After Customization

Figure 5-5: MIT FamilyNet

5.3 "Continue" Conference Management

5.3.1 Original Website

Continue[40] is a free conference management system developed by a team of researchers at Brown University. It allows conference chairs to invite reviewers, authors to submit and track paper submissions, and the program committee to accept or reject papers. The tool supports the entire conference workflow from forming of the program committee all the way up to the acceptance and rejection of each paper. The default behavior is as follows:

- Before submission is open to the general public, conference chair can invite people to join the program committee.
- During the submission phase, authors can upload one or more papers.
- During the bidding phase, chairs and reviewers submit bids for papers they're interested in.
- During the assignment phase, chairs assign papers to reviewers based on their bids.
- During the reviewing phase, reviewers submit reviews for papers they are assigned to.
- During the discussion phase, the program committee read the reviews and decide which paper to accept.
- During the notification phase, authors are informed of the decisions and can read anonymized reviews.

However, there are many corner cases left unspecified above. For example, if a committee member submitted a paper to his own conference, can he find out who reviewed his paper? The rule above describing the discussion phase seems to suggest every committee member can see the full text and authorship information of every review, whereas the notification rule clearly intends to keep the identities of reviewers from authors. Due to the fact that its access policy is tricky and very state-dependent, its developers decided to write an Alloy model to analyze it.

These conflicts of interest arise very naturally whenever a user can belong to multiple conceptual roles in the same model, such as in the case of a conference management website. If a strictly role-based access policy language is used for describing the policy, the same user may get assigned conflicting capabilities where the final decision to grant a request or not depends on technicalities such as the ordering of the rules.

5.3.2 Weballoy Implementation

In attempting to model the access policy of Continue using Weballoy, the following assumptions were made about conflict resolution:

- A reviewer cannot submit a bid (except "Conflict") for his own paper.
- A chair cannot see the bids nor make the reviewer assignment for his own paper.
- At no time can a author see the bids, assignments, and reviewer names of his own paper, even if he's a committee member.

The data model (Figure 5-7) consists of 4 enumerations and 6 user-defined signatures (`User`, `Conf`, `Paper`, `Bid`, `Review`, and `Comment`). Since each `Bid` is simply an owner (inherited from the `Entity` signature) and the bid type, it could have been modeled as a ternary field in the `Paper` signature; however, since the entire field values of a particular atom are either visible to a user or not, it would not be possible to selectively hide certain tuples while showing others. Making each bid a full atom provides that flexibility.

Some of the "Continue" rules are naturally modeled as consistency constraints (Figure 5-7). For example, the rule stating that only assigned reviewers can submit reviews can be expressed by saying the owners of `Review` atoms must be a subset of the set of assigned reviewers for that paper. Likewise, since bids, reviews, and comments are represented by atoms of signatures rather than as tuples, the requirement that a reviewer cannot submit two separate reviews for the same paper can be expressed by saying the number of distinct `Review` atoms is equal to the number of distinct owners of `Review` atoms.

Most of the rules, however, are concerned with the act of tuple insertion and deletions. While they can be modeled using pre-conditions or post-conditions, each field would require two predicates: one for insertion and one for deletion. Fortunately, these rules can be described much more concisely using the capability approach (shown in Figure 5-8) and by appropriate use of helper functions such as "`at`" and "`my`" (defined in Figure 5-9). For example, the following capability expression (if not disallowed by a pre-condition, post-condition, or consistency constraint) allows a reviewer, who has created a review, to read and write the contents of that review:

```
    RW -> pcOf.at[Reviewing].papers.reviews.my -> (Review$.fields)
```

The helper function "`at`" is applied to the result of "`pcOf`" and selects the conferences that are currently in the reviewing phase. Relational navigation extracts the reviews associated with papers in those conferences, then filters them for reviews created by the current user. The "`RW`" function evaluates to "`R+A+D`" thereby granting the rights for reading, addition, and removal in the same rule.

Two post conditions are defined (Figure 5-10) to ensure a newly created conference or paper does not start with illegal values.

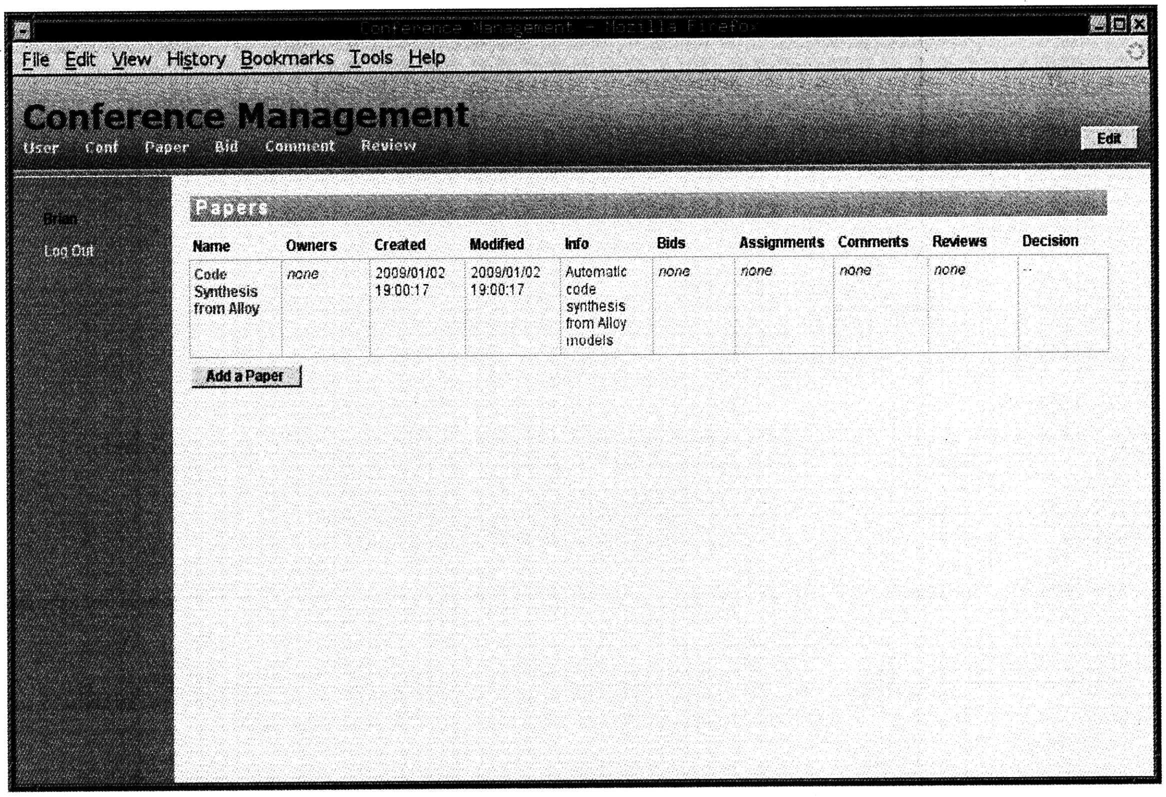
Since each `Bid` atom is defined to belong to exactly one `Paper` by a consistency constraint, removing a bid from a paper's `bids` list must always be accompanied by the destruction of that `Bid` atom in the same transaction. This is an unnecessary complications for the front-end. To avoid this, four triggers are defined (Figure 5-10) to trigger the automatic destruction of a `Bid`, `Review`, `Comment`, or `Paper` atom wherever it is removed from its associated parent `Paper` or `Conf` atom. (There is no need to add triggers for the reverse situation where a `Bid` atom is destroyed without first removing it from a paper's `bids` field, since the semantics defined for Weballoy states that any tuple containing a destroyed atom is automatically removed.)

5.3.3 Limitations

Besides missing the file upload feature, the automatically generated website also lacks an interesting feature of Continue: the ability to generate a one-time URL that grants the visitor temporary permission to perform certain tasks.

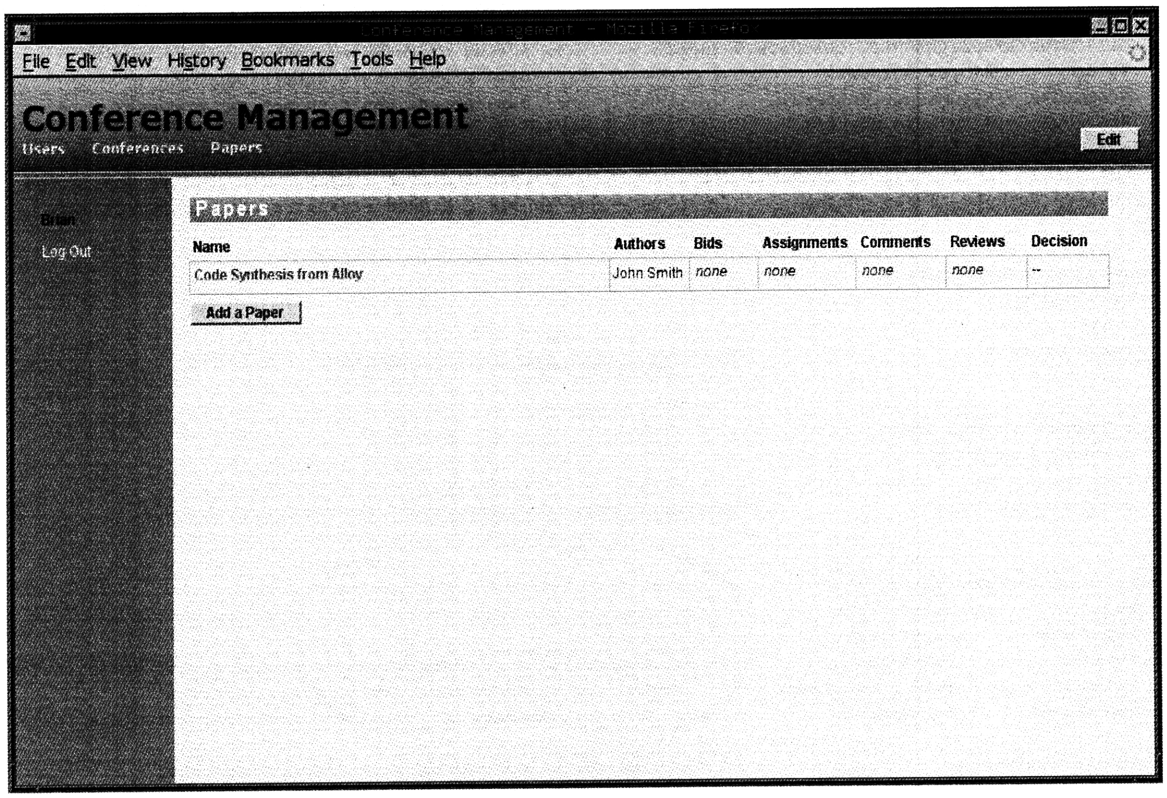
As a convenience, users invited to contribute comments to a paper do not have to login to the website; instead, Continue can generate a URL that embeds the necessary credentials for adding comments. Once the user visited the URL and submitted a comment, the one-time URL is invalidated and cannot be used again.

To emulate this feature using Weballoy, a new anchor (Table 4.1) can be introduced that encodes a session cookie in the URL. It is straightforward to provide additional syntax for modifying the binary relation `String->LoginUser` (Section 3.5) implicitly maintained by the server, so that users visiting the given URL will be automatically associated with a particular `LoginUser` atom. However, the current session retention policy of Weballoy allows a cookie to be repeatedly used. The policy will need to distinguish between cookies that persist until logout and cookies that should be invalidated after one use.


Conference Management - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Conference Management
User Conf Paper Bid Comment Review
Edit
Brian
Log Out
Papers
Add a Paper

Name	Owners	Created	Modified	Info	Bids	Assignments	Comments	Reviews	Decision
Code Synthesis from Alloy	*none*	2009/01/02 19:00:17	2009/01/02 19:00:17	Automatic code synthesis from Alloy models	*none*	*none*	*none*	*none*	--

Before Customization


Conference Management - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Conference Management
Users Conferences Papers
Edit
Brian
Log Out
Papers
Add a Paper

Name	Authors	Bids	Assignments	Comments	Reviews	Decision
Code Synthesis from Alloy	John Smith	*none*	*none*	*none*	*none*	--

After Customization

Figure 5-6: Conference Submission Managemenet

```
enum Phase      { Init, PreSubmit, Submit, Bidding, Assigning, Reviewing, Discuss, Notify, Publish }

enum BidType    { LoveTo, CanReview, NoPreference, DontWantTo, Conflicted }

enum Score      { ScoreA, ScoreB, ScoreC, ScoreD }

enum Expertise { X, Y, Z }

sig User extends LoginUser {
  info         : LongString,
  chairOf      = Conf  & (@owners.this),
  reviewerOf   = reviewers.this,
  submitted    = Paper & (@owners.this),
  submittedTo = papers . (@owners.this)
}

sig Conf extends Entity {      // the inherited "owners" field represents the conference chairs
  info      : LongString,      // can be modified at any time by the conference chair(s)
  phase     : Phase,           // can be advanced at any time by the conference chair(s)
  reviewers : set User,        // can be added/deleted during PreSubmission
  papers    : set Paper        // can be added/modified during Submission and Notify
} {
  disj[owners, reviewers]      // The same person cannot be chair and reviewer for the same conference
}

sig Paper extends Entity {     // the inherited "owners" field represents the paper author
  info         : LongString,   // this represents the paper itself (until file uploads are implemented)
  bids         : set Bid,      // the "owners" field of a Big atom stores the bidder
  assignments : set User,
  comments     : set Comment,  // the "owners" field of a Comment atom stores the comment author
  reviews      : set Review,   // the "owners" field of a Review atom stores the review author
  decision     : lone Bool     // this field is empty when a decision has not been made
} {
  one papers.this                                  // Every paper is in exactly one conference
  disj[owners, assignments]                        // Author cannot be a reviewer for the paper
  disj[owners+assignments, comments.@owners]       // Author or reviewer cannot submit comments
  #bids     = #(bids.@owners)                      // Same person can't submit two bids
  #comments = #(comments.@owners)                  // Same person can't submit two comments
  #reviews  = #(reviews.@owners)                   // Same person can't submit two reviews
  assignments in (papers.this).(@owners+reviewers) // Only PC members can be assigned to review
  Conflicted !in @owners.assignments.bid           // Cannot assign someone whose bid is Conflicted
  reviews.@owners in assignments                   // Only assigned reviewers can submit reviews
}

sig Bid extends Entity {
  bid: BidType
} {
  one bids.this                  // Every bid is associated with exactly one paper
}

sig Comment extends Entity {
  comment: LongString
} {
  one comments.this              // Every comment is associated with exactly one paper
}

sig Review extends Entity {
  score         : Score,
  expertise     : Expertise,
  review        : LongString,        // This text will be readable by paper author during notification
  privateReview : lone LongString,   // This text is intended only for committee members
  subreviewers  : set String         // This stores the names of people who contributed to this review
} {
  one reviews.this                   // Every review is associated with exactly one paper
}
```

Figure 5-7: Data model and Consistency Constraints

```
fun policy : Action->univ->univ
{
 // Everyone can create a conference, paper, bid, comment, or review object
 ADD -> (Conf$ + Paper$ + Bid$ + Comment$ + Review$)

 // Everyone can see every conference and even its list of Paper atoms.
 // However, the fields of a paper (owner, reviewers, decision...) are controlled separately.
 // So, in terms of Conf$papers, this rule merely allows everyone to know the number of submissions.
 + R  ->  Conf  ->  field$

 // Chairs can modify the basic information and phase of a conference.
 // "pred phase.preAdd" shown later will enforce that the phase is advanced step by step
 + W  ->  me.chairOf  ->  (Conf$info + Conf$phase)

 // Chairs can add/delete chairs during the Init phase, and add/delete reviews during PreSubmit phase
 + W  ->  me.chairOf.at[Init]      ->  Entity$owners
 + W  ->  me.chairOf.at[PreSubmit] ->  Conf$reviewers

 // Everyone can submit papers during the Submit phase
 + A  ->  Conf.at[Submit]  ->  Conf$papers

 //-------------------------------------------------------------------------------------------//

 // Users can modify his own information
 + RW  -> me  ->  (User$info + LoginUser$email + LoginUser$password)

 // Everyone can see chairs' and reviewers' basic information
 + R   -> (Conf.owners + Conf.reviewers)  ->   User$info

 // PC members can see authors' basic information and emails
 + R   -> pcOf.papers.owners   ->  (User$info + LoginUser$email)

 // PC members can enumerate the list of Bid/Review/Comment atoms (except for his own papers)
 // The actual bid/review/comment details are controlled by their own fields.
 // So this rule on its own merely allows PC members to know the number of bids/reviews/comments.
 + R   -> (pcOf.papers - my[Paper])  ->  field$

 //-------------------------------------------------------------------------------------------//

 // Author can read his own name; author can modify paper itself during the Submit and Notify phases
 + R   -> my[Paper]                   -> (Paper$info + Entity$owners)
 + W   -> my[Paper].at[Submit+Notify]  ->  Paper$info

 // PC member can submit bids during the Bidding phase
 + A   -> pcOf.at[Bidding].papers          -> (Paper$bids)
 + RW  -> pcOf.at[Bidding].papers.bids.my  -> (Bid$.fields)

 // Chair can see the bid details (except of his own paper)
 + R   -> (me.chairOf.papers - my[Paper]).bids -> field$

 // Chair can assign papers to reviewers (other than his owner papers) during the Assigning phase
 + W   -> (me.chairOf.at[Assigning].papers - my[Paper]) ->  Paper$assignments

 // PC member can submit reviews and comments during the Reviewing phase
 + A   -> pcOf.at[Reviewing].papers              -> (Paper$reviews + Paper$comments)
 + RW  -> pcOf.at[Reviewing].papers.reviews.my  -> (Review$.fields)
 + RW  -> pcOf.at[Reviewing].papers.comments.my -> (Comment$.fields)

 // PC member can read all reviews and comments (except his own paper)
 + R   -> (pcOf.papers - my[Paper]).reviews  ->  field$
 + R   -> (pcOf.papers - my[Paper]).comments ->  field$

 // Chair can accept/reject paper (other than his own) during the Discuss phase
 + W   -> (me.chairOf.at[Discuss].papers - my[Paper]) ->  Paper$decision

 // Author can read the decision, review score, and review text during Notify and Publish phases
 + R   -> my[Paper].at[Notify+Publish]         -> (Paper$decision + Paper$reviews)
 + R   -> my[Paper].at[Notify+Publish].reviews -> (Review$score + Review$review)
}
```

Figure 5-8: Capabilities

```
// Returns the conferences that the current user is a chair or reviewer of.
fun pcOf: set Conf  { me.chairOf + me.reviewerOf }

// Returns the subset of "someConfs" whose phase is one of the phase listed in itsPhase
fun at[someConfs: Conf, itsPhase: Phase]: set Conf  { {c: someConfs  | c.phase in itsPhase} }

// Returns the subset of "somePapers" whose phase is one of the phase listed in itsPhase
fun at[somePapers: Paper, itsPhase: Phase]: set Paper  { {p: somePapers | (papers.p).phase in itsPhase} }

// Returns the subset of "entities" whose owner is the current user
fun my[entities: Entity] : set entities  { {e: entities | me in e.owners} }
```

Figure 5-9: Helper functions

```
// Ensures the phase strictly advances, and ensures minimum necessary conditions for phase advancement
pred phase.preAdd {
  let conf=this.univ,  new=univ.this,  old=conf.phase,  papers=conf.papers,  reviewers=conf.reviewers {
    old = new.prev
    old = Bidding   => (all p: papers | some p.bids) && (reviewers in papers.bids.owners)
    old = Assigning => all p: papers | some p.assignments
    old = Reviewing => all p: papers | some p.reviews
    old = Discuss   => all p: papers | some p.decision
  }
}

// Ensures a conference is created in the Init phase with no papers and no reviewers
pred Conf.postAdd { this.phase=Init  &&  no this.papers  &&  no this.reviewers }

// Ensures a paper is created with no bids/assignments/comments/reviews/decision
pred Paper.postAdd { no (this.bids + this.assignments + this.comments + this.reviews + this.decision) }

// This simplifies the user interface by deleting a Paper automatically when it's removed from a conference
fun papers.onDel: Paper { univ.this }

// This simplifies the user interface by deleting a Bid automatically when it's removed from a Paper
fun bids.onDel: Bid { univ.this }

// This simplifies the user interface by deleting a Review automatically when it's removed from a Paper
fun reviews.onDel: Review { univ.this }

// This simplifies the user interface by deleting a Comment automatically when it's removed from a Paper
fun comments.onDel: Comment { univ.this }
```

Figure 5-10: Pre-conditions, post-conditions, and triggers

Chapter 6

Conclusion

6.1 Discussion

When this research project first began in 2007, there were already many mature competing approaches to access policy specification, validation, website construction, code synthesis, and user interface construction. Those tools offered an impressive array of features and capabilities for addressing one or two specific aspects, but combining them to form a policy-rich website required extensive manual efforts by website designers, and the guarantees offered by one tool could be invalidated by the use of another.

Initially I considered extending an existing state-of-the-art system as a starting point. Such a system would already have extensive tool support and a large user community, reducing the amount of implementation required to get started. However, this idea was rejected because I needed to be able to quickly experiment with drastically different approaches and an existing mature implementation would be difficult to modify and extend. Furthermore, most existing systems had a preset operating paradigm tailored for a specific aspect of the problem and would not be suitable for describing and constructing an entire website.

Instead, I attempted to model different aspects of a website using Alloy because it was a very simple general purpose modeling language I was very familiar with, and because it had recently been used successfully [55] by others in my research group to model information flow in cryptographic systems. I have often found it enlightening to model a problem in Alloy before attempting implementation; in this case, it was a bonus that the very same models used for enhancing my understanding became part of the proposed solution.

Since Alloy does not natively support state mutation or provide any built-in mechanisms for describing user actions or intentions, I had to decide on a way to model those aspects. It was tempting to simply add specialized syntax or features for each, creating a new language similar to Alloy but with hardwired special semantics. However, that would require customized versions of the Alloy tool chain and reduce the general applicability of the system. So instead I took the view that the deficiencies should be addressed by generalizing the missing features then adding them as fundamental enhancements to the Alloy language.

As a result, the existing Alloy parser, type checker, analysis engine, evaluator, instance visualizer, and interactive development environment were all used directly for this research. For Weballoy users, this means the Alloy Analyzer can be used as-is for visualizing the data model, generating use cases, and validating the consistency constraints and access policy rules. For Alloy users, this research resulted in enhancements such as the new `String` signature, enumeration, and metamodel capabilities. The new relational data store and its optimized evaluator are being incorporated into the standard Alloy Analyzer, and the Weballoy auto-generated user interface may be adapted to serve as an interactive instance editor – something currently missing in the Alloy Analyzer.

Using the same expressive, declarative language surprisingly solved many problems. Models of websites are more concise because the metamodel corresponding to the data model provides the building blocks for specifying access policies. Furthermore, the use of a single unifying language enables a critical separation of concerns allowing each aspect to be addressed individually but still effortless to combine them to form a complete description of an entire website.

The presentation template was the exception: the presentation layer was naturally decoupled, so it was very straightforward to separate out the template from the rest of the Alloy model. Since website developers were already familiar with HTML, CSS, and client-side JavaScript, I decided to design the template language as a minimal extension of HTML: only three new elements were needed (`<view/>`, `<f:field/>`, and `<f:s:sig/>`).

In the end, the Weballoy tool chain is a proof-of-concept for a unified declarative approach to website modeling and construction. The ideas presented in this thesis augment rather than compete with state-of-the-art systems in each aspect addressed by this work. There are many opportunities for synergy with existing tools:

- Since the Alloy Analyzer can output the metamodel as an XML file, website designers can also validate the data model by writing XSD definitions then using existing XML validation tools.

- The semantically rich transaction format can be parsed by PHP or Java code then checked by another policy checker instead of or in conjunction with the checker generated by Weballoy.

- Suitable XML-RPC libraries can enable existing template engines to read data from Weballoy-generated data stores and allow the vast number of existing artistic templates to be reused. Standard optimizations such as request pipelining (asynchronously issuing the next query without waiting for the last reply) and request buffering (combining independent queries into a single query) could easily be included to minimize the latency of querying and displaying each value on a web page.

In conclusion, many access policy languages support rich declarative specifications, and domain-specific tools exist for efficient code generation. This research shows it is possible to combine them and automatically synthesize a dynamic website with rich access policies from a concise declarative model. The generated system provides an end-to-end guarantee with the code running in the web browsers, the messaging protocols, the access control system, and the code accessing the database all mechanically generated from a checkable high-level model. The technique relies on dual interpretations of the same model. By attaching special semantics to functions with certain names rather than modeling each operation directly, the model is static from Alloy's perspective, but Weballoy can interpret the model as a description of dynamic website behavior. While more research is needed to support the diverse range of commercial websites, I believe this declarative approach provides significant advantages over other techniques for specification, validation, and construction of certain classes of online information systems.

6.2 Future Work

6.2.1 SQL

Using an off-the-shelf SQL relational database in place of the custom tuple store would have many advantages such as greater scalability, data replication, and parallel processing. Furthermore, the generated website could then interface with existing enterprise business applications by communicating via a shared database.

Generating efficient SQL queries is nontrivial and would require adapting SQL query optimization techniques in the context of Alloy expressions.

6.2.2 Optimization

Within the current implementation, there are many places where further optimization is possible. For example, many consistency constraints take the form `(all x:X | ...)` where `X` is a signature and the body of the formula depends on only `x`'s field values without depending on other `X` atoms. That means when an atom is created or modified, it cannot cause field values of other atoms to become inconsistent. A conservative data flow analysis may be able to determine whether this constraint should be checked for every atom every time or whether it suffices to consider only atoms whose field values just changed.

Another optimization opportunity involves pre- and post-conditions. For example, given the following post-condition `postAdd{this.x in y}` and multiple tuples t_1 through t_n, instead of evaluating each "t_i`.x in y`" separately, it may be more efficient (when semantically equivalent) to union t_1 through t_n into a single set t and then evaluate just one formula "`t.x in y`".

6.2.3 Common Idioms

The Weballoy library module contains the `LoginUser` signature for handling the standard user log-in features. The Weballoy compiler recognizes this special signature and automatically provides cookie-based session management with log-in and log-out buttons.

Other common website features such as calendars, contact lists, and forum postings should be added to the Weballoy library module with suitable default access policy rules pre-defined for convenience.

6.2.4 Expressiveness

The pre- and post-conditions, expansion triggers, and capability rules are simple yet expressive enough for many websites. However, certain advanced features such as history-based access control rules and transparent access logging are cumbersome or impossible to express. For example, a rule that grants or denies a request based on the number of previous unsuccessful attempts is currently not expressible in Weballoy since a rejected request, by definition, does not modify the visible database state and the Alloy model currently has no way to refer to external data source such as the access log. Further research is needed to augment the existing access control primitives while retaining the current clean relational semantics.

6.2.5 WYSIWYG Customization

Even though the Weballoy template language is expressive and supports the use of arbitrary HTML, utilizing it currently requires editing an XML file. Use of a graphical XML editor or a suitable plug-in to a graphical HTML editor can reduce the required effort, but it is still overkill for simple customizations such as changing the color or font size on a page.

Ajax-based systems such as Exhibit[12] and DabbleDB[6] allow users to customize some aspect of the presentation by clicking buttons and selecting alternatives from a menu. Incorporating such a feature into Weballoy's user interface and ensuring the customizations persist would significantly reduce the need for manual template editing.

6.2.6 Counterexample-driven Policy Refinement

The generated back-end can record all occurrences of policy violation as well as successful attempts. This log can be analyzed afterwards to validate whether the policy rules written in Alloy correctly reflect the website designer's intentions.

If a request is needlessly denied, the log clearly shows which rule causes the rejection and therefore needs revision. However, it is often unclear how to modify the policy rules if an undesirable request is granted by mistake.

Heuristics may be developed for automatically suggesting a suitable refinement to the policy based on the concrete violation. To verify that the refinement preserves the behavior for unaffected use cases, the server log can be replayed against the old and new policies, or semantic differencing algorithms such as those used by Margrave[16] might be used.

Bibliography

[1] The Alloy Analyzer. http://alloy.mit.edu/.

[2] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained, The Model Driven Architecture: Practice and Promise.* Addison-Wesley, 2003.

[3] Moritz Y. Becker and Sebastian Nanz. A Logic for State-Modifying Authorization Policies. In *In: European Symposium on Research in Computer Security*, 2007.

[4] T. Berners-Lee, L. Masinter, and M. McCahill. RFC 1738: Uniform Resource Locators (URL), 1994.

[5] Hiawatha Bray. "Payroll Website Still Not Secured". *The Boston Globe*, March 1, 2005.

[6] Dabble DB. http://www.dabbledb.com/.

[7] Daniel Jackson. Automating First-Order Relational Logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering.* ACM Press, 2000.

[8] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A Micromodularity Mechanism. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '01)*, 2001.

[9] Django: the Web Framework for Perfectionists With Deadlines. http://www.djangoproject.com/.

[10] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and Reasoning About Dynamic Access-Control Policies. In *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[11] Drupal: an Open Source Content Management Platform. http://www.drupal.org/.

[12] Exhibit. http://simile.mit.edu/exhibit/.

[13] MIT FamilyNet. http://familynet.mit.edu/.

[14] David Ferraiolo and D. Richard Kuhn. Role-Based Access Controls. *15th National Computer Security Conference*, pages 554–563, 1992.

[15] Ronaldo Rodrigues Ferreira. Automatic Code Generation and Solution Estimate for Object-Oriented Embedded Software. In *OOPSLA '08: Proceedings of the 23th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2008.

[16] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and Change-Impact Analysis of Access-Control Policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, New York, NY, USA, 2005. ACM.

[17] Flapjax. http://www.flapjax-lang.org/.

[18] Sonia Flores, Salvador Lucas, and Alicia Villanueva. Formal Verification of Websites. *Electron. Notes Theor. Comput. Sci.*, 200(3):103–118, 2008.

[19] Joan Fons, Vicente Pelechano, and Oscar Pastor. Extending an OO Method to Develop Web Applications. In *12th International World Wide Web Conference*, Budapest, Hungary, 2003.

[20] Theophilos Giannakopoulos, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Welding Alloy Specifications to Adequate Implementations. In *ICSE '09: Proceedings of the 31th International Conference on Software Engineering*, New York, NY, USA, 2009. ACM Press.

[21] Google Health. https://www.google.com/health/.

[22] Google Web Toolkit. http://code.google.com/webtoolkit/.

[23] Ali Nasrat Haidar and Ali E. Abdallah. Towards a Formal Framework for Developing Secure Web Services. *Automated Specification and Verification of Web Systems, 2006. WWV '06. 2nd International Workshop on*, pages 61–70, Nov. 2006.

[24] Hibernate: an open source java persistence framework. http://www.hibernate.org/.

[25] Graham Hughes, Tevfik Bultan, and Muath Alkhalaf. Client and Server Verification for Web Services Using Interface Grammars. In *TAV-WEB '08: Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, pages 40–46, New York, NY, USA, 2008. ACM.

[26] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.

[27] Java Platform Enterprise Edition (Java EE). http://java.sun.com/javaee/.

[28] Jonathan Edwards, Daniel Jackson, and Emina Torlak. A Type System for Object Models. In *Foundations of Software Engineering, Newport, CA*, 2004.

[29] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.

[30] jQuery. http://www.jquery.com/.

[31] JavaServer Faces. http://java.sun.com/j2ee/javaserverfaces/.

[32] JavaServer Pages. http://java.sun.com/products/jsp/.

[33] Eunsuk Kang and Daniel Jackson. Formal Modeling and Analysis of a Flash Filesystem in Alloy. In Egon Brger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2008.

[34] Martin Karusseit and Tiziana Margaria. A Web-Based Runtime-Reconfigurable Role Management Service. *Automated Specification and Verification of Web Systems, 2006. WWV '06. 2nd International Workshop on*, pages 53–60, Nov. 2006.

[35] Martin Karusseit, Tiziana Margaria, and Holger Willebrandt. Policy Expression and Checking in XACML, WS-Policies, and the jABC. In *TAV-WEB '08: Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, pages 20–26, New York, NY, USA, 2008. ACM.

[36] Haim Kilov. From Semantic to Object-Oriented Data Modeling. In *First International Conference on System Integration*, pages 385–393, 1990.

[37] Alexander Knapp and Stephan Merz. Model Checking and Code Generation for UML State Machines and Collaborations. In *FM-TOOLS 2002: 5th Workshop on Tools for System Design and Verification*, 2002.

[38] Kodkod: Solving Formulas with Partial Instances. http://sdg.csail.mit.edu/projects/kodkod.html.

[39] S Kripke. Semantical Analysis of Modal Logic. *Z. Math. Log. Grund. Math. 9*, 1963.

[40] Shriram Krishnamurthi. The CONTINUE Server (or, How I Administered PADL 2002 and 2003). In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 2–16, London, UK, 2003. Springer-Verlag.

[41] Shriram Krishnamurthi, Daniel Dougherty, Kathi Fisler, and Daniel Yoo. Alchemy: Transmuting Base Alloy Specifications into Implementations. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, New York, NY, USA, 2008. ACM Press.

[42] Shriram Krishnamurthi, Robert Bruce Findler, Paul Graunke, and Matthias Felleisen. Modeling Web Interactions and Errors. In *In Interactive Computation: The New Paradigm*. Springer Verlag, 2006.

[43] Daniel R. Licata and Shriram Krishnamurthi. Verifying Interactive Web Programs. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 164–173, Washington, DC, USA, 2004. IEEE Computer Society.

[44] D. Marinov and S. Khurshid. TestEra: a Novel Framework for Automated Testing of Java Programs. In *ASE '2001: 16th IEEE International Conference on Automated Software Engineering*, pages 22–31, 2001.

[45] Microsoft Health Vault. http://www.healthvault.com/.

[46] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[47] phpMyAdmin. http://www.phpmyadmin.net/.

[48] PHPTempalte Theme Engine. http://drupal.org/phptemplate.

[49] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symp. Foundations of Computer Science*, 1977.

[50] Lukas Renggli. Magritte: Meta-Described Web Application Development. Master's thesis, University of Bern, 2006.

[51] Ruby on Rails. http://www.rubyonrails.org/.

[52] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.

[53] Roger Stone. A Lightweight Web GUI Specification and Realisation System and Its Impact on Accessibility. *Automated Specification and Verification of Web Systems, 2006. WWV '06. 2nd International Workshop on*, pages 37–44, Nov. 2006.

[54] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding Minimal Unsatisfiable Cores of Declarative Specifications. In Jorge Cullar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

[55] Emina Torlak, Marten van Dijk, Blaise Gassend, Daniel Jackson, and Srinivas Devadas. Knowledge Flow Analysis for Security Protocols. Technical Report MIT-CSAIL-TR-2005-066, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2005.

[56] Trusted Computer System Evaluation Criteria (DoD Standard 5200.28-STD). *United States Department of Defense*, 1985.

[57] Michael Carl Tschantz and Shriram Krishnamurthi. Towards Reasonability Properties for Access-Control Policy Languages. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 160–169, New York, NY, USA, 2006. ACM.

[58] OASIS eXtensible Access Control Markup Language (XACML) Version 2.0, 2005. http://docs.oasis-open.org/xacml/2.0/.

[59] XForms 1.1. http://www.w3.org/TR/xforms11/.

[60] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.

[61] XSL Transformation (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/.

[62] Tom Zeller. "Not Yet in Business School, and Already Flunking Ethics". *The New York Times*, March 14, 2005.