Rubicon: Bounded Verification of Web Applications

Rubicon: Bounded Verification of Web Applications

Joseph P. Near, Daniel Jackson
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA, USA
{jnear,dnj}@csail.mit.edu

ABSTRACT

We present Rubicon, an application of lightweight formal methods to web programming. Rubicon provides an embedded domain-specific language for writing formal specifications of web applications and performs automatic bounded checking that those specifications hold. Rubicon's language is based on the RSpec testing framework, and is designed to be both powerful and familiar to programmers experienced in testing. Rubicon's analysis leverages the standard Ruby interpreter to perform symbolic execution, generating verification conditions that Rubicon discharges using the Alloy Analyzer. We have tested Rubicon's scalability on five real-world applications, and found a previously unknown security bug in Fat Free CRM, a popular customer relationship management system.

Keywords

Formal methods, programming languages, web programming

1. INTRODUCTION

Web applications have experienced explosive growth over recent years. Testing methodologies for web applications are beginning to catch up, but despite their utility in building correct programs, there has been as yet little use of formal methods in the domain.

This paper introduces Rubicon, an application of lightweight formal methods to web programming. Rubicon allows programmers to write specifications of the behavior of their web application and performs automatic bounded analysis to check those specifications against the implementation. Rubicon aims to be both powerful and easy to use: its specification language is expressive but based on a domain-specific language for testing.

Rubicon is implemented as a library for the Ruby programming language, and targets applications written using the popular Rails web programming framework. Rubicon's specification language is an extension of the Ruby-based RSpec domain-specific language for testing [7]; Rubicon adds the quantifiers of first-order logic, allowing programmers to replace RSpec tests over a set of mock objects with general specifications over all objects. This compatibility with the existing RSpec language makes it easy for programmers already familiar with testing to write specifications, and to convert existing RSpec tests into specifications.

Rubicon's automated analysis comprises two parts: first, Rubicon generates verification conditions based on specifications; second, Rubicon invokes a constraint solver to check those conditions. The Rubicon library modifies the environment so that executing a specification performs symbolic execution, producing verification conditions rather than values. To check the verification conditions, Rubicon currently compiles them into Alloy [14], a lightweight specification language for software design, for analysis using the Alloy Analyzer, a fully-automatic bounded analysis engine for Alloy. Alloy's relational language is convenient because its semantics closely match those of relational databases, but in principle, an unbounded SMT solver or theorem prover could be used in place of Alloy.

We tested Rubicon's performance on five open-source web applications for which the original developers have written RSpec tests. We took a random sample of these tests and converted them into Rubicon specifications; in every case, Rubicon's analysis took no more than a few seconds per specification. In the largest of these applications, a customer relationship management system called Fat Free CRM, Rubicon's analysis uncovered a previously unknown security bug. The authors of Fat Free CRM have acknowledged this bug and are preparing a fix.

We have released Rubicon under the terms of the GPL at: `http://people.csail.mit.edu/jnear/rubicon`.

The contributions of this paper include:

- Rubicon, a domain-specific language for expressing specifications of web applications;
- a scalable, automatic, symbolic-execution-based bounded checker for Rubicon specifications;
- an evaluation of Rubicon on five applications, and the discovery of a previously unknown security bug.

```
1  describe UsersController do
2    it "should expose the requested user as @user and
          render [show] template" do
3      @user = Factory(:user)
4      get :show, :id => @user.id
5      assigns [:user].should == @user
6      response.should render_template("users/show")
7    end
8  end
9
10 describe ContactsController do
11   it "should redirect to contact index if the contact
          is protected" do
12     @private = Factory(:contact, :user =>
            Factory(:user), :access => "Private")
13     get :show, :id => @private.id
14     flash [:warning].should_not == nil
15     response.should redirect_to (contacts_path)
16   end
17 end
```

Figure 1: RSpec Tests for Displaying Users and Restricting Access to Private Contacts

2. THE RUBICON LANGUAGE

Rubicon provides an embedded domain-specific language for writing specifications. This language is based on RSpec [7], a testing framework for the Ruby language whose goal is to make testing easier and more useful, and that has made test-driven design popular amongst Ruby programmers. RSpec tests are concise, avoid repetition, and resemble English specifications of application features. The framework also encourages programmers to write documentation for each test by providing fields for that documentation and using it to generate error reports when tests fail.

The Rubicon language is designed for writing specifications of Rails applications. Rails is the most popular web programming framework for Ruby, and was designed specifically to allow testing with RSpec. The integration of Rails with RSpec means that test-driven development is just as popular in the Rails community as it is in the larger Ruby community, and many open-source Rails applications are shipped with large RSpec test suites.

2.1 The RSpec Approach

To introduce the style of testing promoted by the RSpec library, we take the open-source Rails application Fat Free CRM[1] as a case study. Fat Free CRM is a customer relationship management system designed to be used to build customized CRM systems for use in organizations. Development of Fat Free CRM began in 2008 and the software is currently maintained by Michael Dvorkin and Nathan Broadbent. Fat Free CRM is released under the AGPL, its codebase comprises 23kloc, but is reasonably simple to understand, and the developers have written more than 1000 RSpec tests.

Figure 1 contains two examples of RSpec tests from the Fat Free CRM codebase. The first (lines 1-8) is intended to test that the show action correctly displays summaries of the site's users. What it actually tests is that a particular user (created by the call to the Factory) is displayed. The test begins with a natural-language specification of the feature under test (line 2); the body of the test (lines 3-7) is Ruby code written in the RSpec domain-specific language. This particular test constructs a mock object representing a single user (line 3), requests the summary page for that user (line 4), and then checks that the user that will be displayed matches the mock object just created (line 5). This check is written as an RSpec assertion, the general form of which is *a.should p b*, where *a* and *b* are objects and *p* is a predicate describing the desired relationship between them. The *assigns* variable is actually a Ruby hash populated by the Rails framework to contain the names and values of the instance variables set by the page request in line 4.

The second test (lines 10-17) is intended to check that private contacts are never displayed to users other than their owners. What it actually tests is that a particular private contact is hidden from a particular user. The test begins by building a mock object for the private contact owned by a mock user (line 12). The test then attempts to display the private contact (line 13). Since the mock user created as the owner of the contact in line 12 is by default distinct from the mock user representing the currently logged-in user, this request should fail. The test checks that it does indeed fail by asserting that the value of flash [:warning], which displays site errors, is populated (line 14), and that the user should be redirected to the index of contacts (line 15).

The example tests in Figure 1 check the desired properties in only a single case; they may easily miss corner cases that the programmer does not anticipate. For example, the second test passes, but the intended property—that the show action in the contacts controller preserves permissions—is actually false! In fact, displaying a contact also causes a summary of the associated opportunities to be displayed; these opportunities also have associated user permissions. The test either ignores the fact that opportunities will be displayed, or assumes that the permissions on these opportunities will match those on the corresponding contact. If they do not, a user may be allowed to see another user's private opportunity. This is an example of a corner case that is unlikely to be caught by testing alone, since testing requires the programmer to specify exactly one case to test—and the programmer is likely to test only the case in which the permissions match.

2.2 Adding Quantifiers

Rubicon allows programmers to write *specifications* for web applications by extending the RSpec language with the quantifiers of first-order logic. Rather than testing just a single case for compliance with the application's specification, Rubicon performs symbolic execution for a truly arbitrary case, generates verification conditions that cover *all* possible cases, and uses an automatic bounded verifier to check them.

Rubicon introduces two new methods on objects—*forall* and *exists*—which represent the universal and existential quantifiers of first-order logic. Both methods accept a single argu-

[1]`http://www.fatfreecrm.com/`

ment: a Ruby block, representing the property over which the quantifier ranges; quantifiers succeed or fail in the same way as other RSpec tests.

```
SpecFile  ::  describe <class> do <Spec∗> end
Spec      ::  it <string> do <Expr∗> end
Expr      ::  <Ruby Expression>
          |   <object>.forall do |x| <Expr> end
          |   <object>.exists do |x| <Expr> end
          |   <class>.forall do |x| <Expr> end
          |   <class>.exists do |x| <Expr> end
          |   <Expr>.implies do <Expr> end
          |   <object>.should_equal <Expr>
          |   <object>.should_not_equal <Expr>
          |   <object>.should <Pred> <Expr>
          |   <object>.should_not <Pred> <Expr>
Pred      ::  ==
          |   be
          |   include
          |   raise
          |   throw
          |   respond_to
          |   have
```

Figure 2: Syntax of Rubicon Specifications

Figure 2 specifies the core syntax of Rubicon specifications. Rubicon's syntax is exactly that of RSpec, with the addition of quantifiers. A set of Rubicon specifications begins with a *describe* block specifying the class being specified; each individual specification is written inside an *it* block with a string documenting that specification. A specification is simply a sequence of expressions, each of which may be a standard Ruby expression or a Rubicon assertion. A quantifier is also an assertion; *a.forall do* $|x|$ *b end* means that the assertions in b should hold for all possible values of x from the set a, and *a.exists do* $|x|$ *b end* means that the assertions in b should hold for at least one value of x from a.

```
1   describe UsersController do
2     it "should expose the requested user as @user and
          render [show] template" do
3       User.forall do |user|
4         get :show, :id => user.id
5         assigns [: user ]. should == user
6         response.should render_template("users/show")
7       end
8     end
9   end
10
11  describe ContactsController do
12    it "should redirect to contact index if the contact
          is protected" do
13      User.forall do |user|
14        Contact.forall do |private|
15          set_current_user(user)
16          get :show, :id => private.id
17          ((private.access == "Private") &
18           (private.user != user)).implies do
19            flash [: warning]. should_not == nil
20            response.should redirect_to (contacts_path)
21          end
22        end
23      end
24    end
25  end
```

Figure 3: Rubicon Specifications for Displaying Users and Restricting Access to Private Contacts

Figure 3 presents Rubicon versions of the RSpec tests from Figure 1. This specification for displaying users (lines 1-9) quantifies over *all* users, rather than testing the intended property for just a single user. The block passed to the *forall* method is precisely the code we wrote in the original test—only the underlined code has changed, reflecting the introduction of quantified variables in place of mock objects.

The specification for restricting access to private contacts (lines 11-25) changes only slightly more. In order to check that permissions are preserved no matter which user is logged in, we quantify over users (line 13) and then set the logged-in user correspondingly (line 15). To express the class of contacts for which the property should hold, we introduce a logical implication (lines 17-18) whose left-hand side requires that the contact's access should be set to private and that its owner should be distinct from the logged-in user.

2.3 The Power of Specification

By writing full specifications of application behavior, programmers can catch errors that tests alone are unlikely to find. Rubicon's automated analysis of formal specifications can explore corner cases, check for general regression errors, and build complicated object hierarchies for testing—areas in which standard RSpec tests fall short. Moreover, Rubicon specifications are often more concise than the RSpec tests they replace, since the programmer may replace complicated code for constructing mock objects with simpler quantifiers.

The previously mentioned fault involving the permissions on opportunities associated with contacts, for example, is unlikely to be caught in testing, since its discovery requires the specific situation in which the user in question has permission to view a particular contact, but does *not* have permission to view the associated opportunity. Because mock objects must take concrete values, a programmer will tend to construct mock objects that represent the most common situation—in this case, one in which the permissions on the contact and its associated opportunities match.

In writing specifications, on the other hand, the tendency is to provide as *little* information as possible, so as to have the highest chance of discovering corner cases that were not considered by the programmer. The Rubicon specification in Figure 4 checks the property that permissions are actually respected when displaying a contact. The specification

```
1  describe ContactsController do
2    it "should not display other users' private contacts
         or opportunities" do
3      User.forall do |u|
4        Contact.forall do |c|
5          Opportunity.forall do |o|
6             set_current_user (u)
7             get :show, :id => c.id
8
9             ((c.access == 'private') &
10             (c.user != u)).implies do
11               assigns[:contact].should_not == c
12            end
13
14            ((o.access == 'private') &
15             (o.user != u)).implies do
16               assigns[:contact].opportunities should_not
17                 include o
18            end
19         end
20       end
21     end
22   end
23 end
```

Figure 4: Rubicon Specification for Displaying Contacts

quantifies over users, contacts, and opportunities (lines 3-5), sets the current user to the quantified one (line 6), and requests the contact display page (line 7). The specification then checks two properties: first, that for *any* contact, if that contact's permissions are set to private and its contact's owner is not the current user, then that contact should *not* be displayed (lines 9-12); and second, that for *any* opportunity, if that opportunity's permissions are set to private and its owner is not the current user, then that opportunity should *not* be included in the list of opportunities to be displayed (lines 14-17).

This specification makes no assumptions about any properties of the contact and opportunities in question; most importantly, it does not rule out the case that the requested contact is public, but one of the associated opportunities is private and not owned by the current user. Indeed, Rubicon catches this case immediately, returning the following counterexample for the second property:

```
=> u = User { :id => 1 ... }
     c = Contact { :access => 'public', :user => u,
                   :opportunities => [o] ... }
     o = Opportunity { :access => 'private',
                       :user => user1 ...}
```

This counterexample represents the case in which the current user owns the contact being displayed, but does *not* own the opportunity associated with that contact. Even though that opportunity is private, it will be displayed to the user.

The blame for this fault lies in the assumption, made in the show action of the contacts controller, that the permissions and ownership of a contact and its opportunities will be identical. The show method uses the my method of the Contact class to determine which contact to display; the developers of Fat Free CRM have redefined the my method elsewhere so as to return only those records that the current user has permission to view. Referencing a particular contact's opportunities field, however, accesses the associated opportunities *directly*, bypassing the redefined my method, and *ignoring* the opportunities' permissions settings. If a user has permission to display a contact, then, he or she will be able to access all of its associated opportunities, regardless of their permission settings.

```
describe ContactsController do
  it "should be able to associate newly created
       contact with the opportunity" do
    @opportunity = Factory(:opportunity, :id => 987);
    @contact = Factory.build(:contact)
    Contact.stub!(:new).and_return(@contact)

    xhr :post, :create, :contact => { :first_name =>
         "Billy"}, :account => {}, :opportunity => 987
    assigns(:contact).opportunities.should
         include(@opportunity)
    response.should render_template("contacts/create")
  end
end
```

Figure 5: RSpec Test for Contact Creation

```
describe ContactsController do
  it "should not associate another user's private
       opportunity with newly created contact" do
    User.forall do |user|
      Contact.forall do |contact|
        Opportunity.forall do |opportunity|
          set_current_user (user)
          xhr :post, :create, :contact =>
                contact.attributes, :opportunity =>
                opportunity.id

          ((opportunity.user != user) &
           (opportunity.access=='private')).implies do
             assigns[:contact].opportunities.should_not
                  include opportunity
          end
        end
      end
    end
  end
end
```

Figure 6: Rubicon Specification for Contact Creation

2.4 Exploiting the Bug

The corner case discovered in the previous section only qualifies as a serious security bug if it is possible to exploit it. The exploit described in the counterexample requires an invariant on the database to be broken—that the permissions of all the opportunities associated with a particular contact are compatible with the permissions on that contact. Rubicon can help us again, this time in checking whether or not it is possible to create a new contact that violates the invariant.

Invariants can be checked the same way using tests, but the contrived nature of test cases makes this strategy less useful than might be hoped—run-time assertions checking for invariant violations are therefore much more popular. Figure 5 contains the RSpec test written by the Fat Free CRM developers to check that the invariant is preserved. As usual, this test uses mock objects (lines 3-5) to construct the common case—one with the default permissions—creates a new contact (line 7), and tests that the opportunity is correctly associated with the contact (line 8).

The corresponding Rubicon specification (Figure 6), on the other hand, quantifies over all users, contacts, and opportunities (lines 3-5), constructs a new contact associated with the quantified opportunity (line 7), and checks that if the opportunity is private and not owned by the current user, then it should not be included in the resulting contact's set of associated opportunities (lines 9-11)—in other words, it should be impossible to create a contact that violates the invariant.

Once again, Rubicon's analysis yields a counterexample, informing us that it is indeed possible to violate the invariant:

```
=> user = User { :id => 1 ... }
    contact = Contact { : opportunities  => [opportunity]
        ... }
    opportunity = Opportunity { :access => 'private',
        : user => user1 }
```

Investigating the code for the `create` action in the contacts controller, we found that the code that looks up the attached opportunity uses the `find` method, rather than the permission-enforcing `my` method, to find the opportunity whose ID is referenced in the HTML form submitted by the user.

Exploiting this security bug, then, is as easy as submitting a contact creation request with the ID of another user's opportunity in the "opportunity_id" HTML field. The developers of Fat Free CRM have acknowledged the bug, and are working on a fix.

3. THE RUBICON ANALYSIS

Rubicon is implemented as a library on top of Ruby, RSpec, and Rails. It extracts verification conditions from specifications by using the standard Ruby interpreter to perform symbolic, rather than concrete, execution; instead of concrete values, this style of execution produces abstract syntax trees representing the appropriate verification conditions. Rubicon then compiles the verification conditions into an Alloy [14], a lightweight specification language for software engineering, and performs bounded analysis using the Alloy Analyzer.

E(obj. forall b)	$\equiv$	$\forall x.$ E(b.call (x))
E(obj. exists b)	$\equiv$	$\exists x.$ E(b.call (x))
E(obj.should p e)	$\equiv$	E(obj.p(e))
E(obj.should_not p e)	$\equiv$	$\neg$E(obj.p(e))
E(e)	$\equiv$	RubyInterpreter (e)

Figure 7: Semantics of Rubicon Specifications

C(obj. forall b)	$\equiv$	*all(x,* C(b.call (x)))
C(obj. exists b)	$\equiv$	*some(x,* C(b.call (x)))
		(x is a new symbolic object)
C(obj.should p e)	$\equiv$	*call(:should*, C(obj), C(p), C(e)*)*
C(obj. attribute)	$\equiv$	*field_ref(* C(obj), C(args)*)*
C(obj.meth(args))	$\equiv$	*call(* meth, C(obj), C(args)*)*
C(obj.where(e))	$\equiv$	*query(* v, obj, C(e)*)*
		(if obj is symbolic)
C(e)	$\equiv$	RubyInterpreter (e)
		(if e is concrete)

Figure 8: Compiling Specifications to Abstract Syntax Trees

3.1 Rubicon's Semantics

Rubicon's semantics are intended to match those of Ruby precisely, and to combine Ruby's semantics in a natural way with the standard semantics of the quantifiers of first-order logic. Figure 7 contains an informal summary of Rubicon's semantics, using the standard quantifier symbols ($\forall, \exists$) and a function representing the semantics of the standard Ruby interpreter (RubyInterpreter). Given standard quantification over Ruby values, we can represent Rubicon's basic assertions (*should* and *should_not*) by simply invoking the predicate on the object in question; a true result means that the assertion holds. It is the goal of Rubicon's implementation to implement these semantics faithfully.

3.2 Rubicon's Implementation

To implement the semantics in Figure 7, Rubicon transforms specifications into verification conditions represented as abstract syntax trees and checks those conditions using a constraint solver. Figure 8 summarizes that transformation; the transformation is based on the use of symbolic objects defined by the Rubicon library to represent quantified variables.

To avoid re-implementing the Ruby interpreter, Rubicon implements the transformation from Figure 8 by persuading the standard Ruby interpreter to perform symbolic execution. Rubicon accomplishes this by defining symbolic objects in such a way that all method invocations on symbolic objects yield abstract syntax trees rather than values. Since specifications necessarily refer to symbolic objects if they use quantifiers, Rubicon can use the results of running code with symbolic objects to build abstract syntax trees representing verification conditions for the specification.

This strategy works especially well for web applications, for two reasons: first, the database schema of a web application specifies exactly the set of possible symbolic objects;

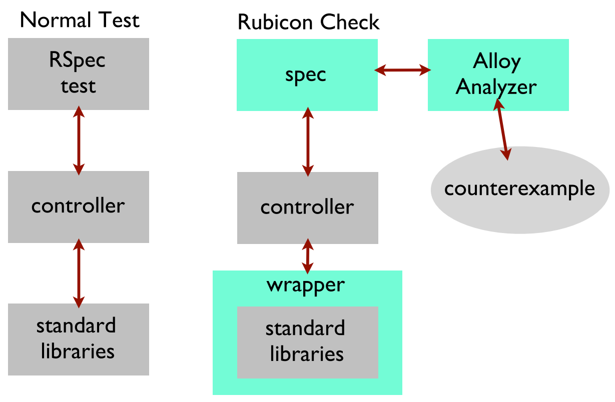

Figure 9: Comparison Between RSpec Execution and Rubicon Analysis

and second, web applications typically move much of the application's logic into the database, so the remaining code has few branches. These two facts lend our approach its scalability: by limiting the number of symbolic objects, we perform as much execution as possible over concrete objects (using the standard Ruby interpreter) and reduce the size of the resulting verification conditions; and because the code contains few branches, the branch-explosion problem that typically plagues symbolic execution does not occur.

A diagram comparing Rubicon's execution model to that of RSpec is presented in Figure 9. Rubicon's analysis proceeds in three parts: first, the Rubicon library stubs the standard libraries and ActiveRecord objects; second, Rubicon runs the specification body in the standard Ruby interpreter, producing verification conditions; and finally, Rubicon compiles those verification conditions for the Alloy Analyzer to check.

3.3 Preprocessing: Stubbing Objects

Rails uses the ActiveRecord class as the basis for its object-relational mapper. In a standard Rails application, every object to be stored in the database is descended from ActiveRecord. To determine the set of classes that should be represented by symbolic objects, then, Rubicon builds a list of all the classes descended from ActiveRecord.

The next step is to *stub* those classes—replacing them with new classes that respond the same way to the operations defined on them, but with different behavior. Rubicon takes each ActiveRecord class and redefines two basic types of methods: class methods that query the database, and instance methods that allow the programmer to get and set the attributes (values to be stored in the database) of an object representing a particular record in the database. For example, the call User.all returns a list of all users, and given a user object, user.id returns that user's ID number.

Rubicon redefines these methods to produce abstract syntax trees rather than values: User.all returns Expr_Call(:all, User), and user.all returns Expr_Field_Ref(user, :id).

The classes that define abstract syntax trees are defined so that methods invoked on them produce new trees reflecting

T(*all(x, e)*)	≡	**all** x: *typeOf(x)* | T(e)
T(*some(x, e)*)	≡	**some** x: *typeOf(x)* | T(e)
T(*field_ref(obj, f)*)	≡	T(obj).f
T(*call(:include, obj, a)*)	≡	T(obj) **in** T(a)
T(*call(:==, obj, a)*)	≡	T(obj) = T(a)
T(*query(name, type, e)*)	≡	{ name: type | T(e) }
T(*call(:should, obj, p, a)*)	≡	T(*call(p, obj, a)*)
T(*v*)	≡	v

Figure 10: Compiling Abstract Syntax Trees to Alloy Specifications

these operations, and the *should* method for building assertions is also redefined to produce abstract syntax trees. As a result, the following Ruby expression:

```
user1 . id . should == user2.id
```

Evaluates to the following abstract syntax tree, instead of a value (Expr_Call represents the constructor of an abstract syntax tree node):

```
Expr_Call (:should, :==, Expr_Field_Ref(user1, :id),
        Expr_Field_Ref(user2, :id))
```

Having prepared the environment this way, running Rubicon specification code in the standard Ruby interpreter yields abstract syntax trees representing verification conditions.

3.4 Postprocessing: Producing Alloy Specifications

All that remains is to translate verification conditions into Alloy specifications. Since our abstract syntax trees are designed for this purpose, doing so is straightforward. We summarize the translation in Figure 10.

The Alloy language includes all of first-order relational logic, plus transitive closure. Quantifiers, therefore, are translated into their analogues in Alloy; field references become relational joins based on Alloy's global relations, database queries become Alloy set comprehensions, and Rubicon assertions turn into logical formulas.

Alloy's universe is made up of uninterpreted atoms, which are divided into sets based on user-defined *signatures*. In addition to designating a set of atoms, signatures may define global *relations*; Alloy specifications are typically based on these global relations, and logical formulas are built from the results of relational joins or membership tests over these relations.

Rubicon uses Alloy's atoms to represent objects. It defines a signature for each type of object that is mentioned in a given specification, with relations to represent the values of the object's fields. Given this representation, a field reference in Ruby is equivalent to a relational join in Alloy. This paradigm is popular in Alloy, so the syntax of relational join is designed to make the representation of a field reference appear similar to the typical syntax in programming languages—*obj.f* means the relational join of the atom *obj* with the global relation *f*, but it also represents the reference to field *f* of object *obj*, given our encoding.

To generate a complete Alloy specification, Rubicon constructs a signature definition for every class referenced in the corresponding Rubicon specification. For each signature, Rubicon consults the application's database schema (encoded in the ActiveRecord) to determine the set of attributes associated with that type, and adds a field definition to the signature for each attribute. For example, the first part of the signature definition for the *User* class from Fat Free CRM is as follows:

```
sig User {
  name: String,
  created_at : Datetime,
  updated_at: Datetime,
  email: String ,
   ...
}
```

Rubicon combines these signature definitions with the translated verification conditions and invokes the Alloy Analyzer to determine whether or not the specification is satisfied. The Alloy Analyzer is a tool for automatic bounded analysis of Alloy specifications; it places finite bounds on the number of each type of atom present in the universe, then reduces the specification to propositional logic. The Analyzer uses a model finder, Kodkod [21], which translates to boolean satisfiability (SAT) using algorithms that optimize relational expressions. This makes Kodkod, and thus Alloy, especially suited to database applications. If a counterexample is found, Kodkod maps the SAT result back to a valuation for the original specification's relations.

Rubicon, in turn, maps the counterexample Alloy produces back to a Ruby object structure, which forms the printed counterexample the user sees when a specification is found not to hold.

3.5 An Example: Contact Permissions

As a complete example of Rubicon's analysis, consider the example of checking that the user has permission to view a contact and its associated opportunities (Figure 4). Figure 11 demonstrates how the second of these two properties is checked.

First, during execution by the Ruby interpreter, the property, along with the Fat Free CRM implementation (in this case, the show method defined in full in Figure **??**), produce a verification condition to be checked by the Alloy Analyzer. The page request (line 12) passes the quantified contact's ID to the show method, where the call to the stubbed version of Contact.my.find (line 2) yields a symbolic record representing precisely the quantified contact. The method returns, having set assigns [: contact] to that symbolic record.

The two conditions on the left-hand side of the implication (line 15) evaluate to expressions involving the quantified opportunity; the condition on the right-hand side of that implication (line 16) evaluates to an expression involving both

```
1  def show
2    @contact = Contact.my.find(params[:id])
3    @stage = Setting. unroll (: opportunity_stage )
4    @comment = Comment.new
5     ...
6  end
7
8  User. forall  do |u|
9    Contact. forall  do |c|
10     Opportunity. forall  do |o|
11       set_current_user (u)
12       get  :show, : id  => c.id
13
14       ((o. access  == 'private') &
15        (o. user  != u)). implies  do
16         assigns [: contact ]. opportunities  should_not
17           include  o
18       end
19     end
20   end
21 end
```

⇒

```
22 Expr_Implies (
23   Expr_And(
24     Expr_Call (:==, Expr_Field_Ref(o,  : access ),
25                    ' private ' ),
26     Expr_Call (:!=,  Expr_Field_Ref(o,  : user ),
27                    u)),
28   Expr_Not(
29     Expr_Call (: include ,
30                Expr_Field_Ref(c,  : opportunities ),
31               o)))
```

⇒

```
32 one sig  string_private  extends String {}
33 sig Opportunity{
34   id : ID, access : String ,  ...,   user : set User
35 }
36 sig Contact{
37   id : ID, access : String ,  ...,  opportunities :  set
         Opportunity
38 }
39 sig User{
40   id : ID, ...
41 }
42
43 check {
44   all  u: User |
45     all  c: Contact |
46       all  o: Opportunity |
47         (o. access  =  string_private  and
48          o. user  != u) implies
49            !(o  in  c. opportunities )
50
51 } for 5
```

Figure 11: Verification Condition and Corresponding Alloy Specification

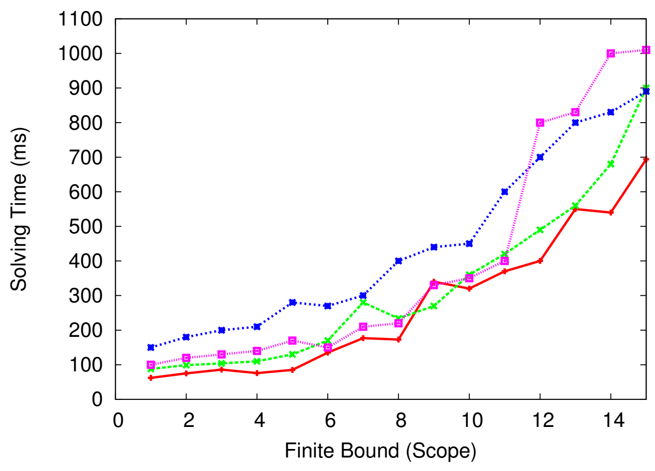

Figure 12: Effect of Finite Bound on Solving Time of Verification Conditions from Examples in Figures 3, 4, and 6

the symbolic record constructed above and the quantified opportunity. The expression representing the complete verification condition is listed in lines 22-31.

Second, the Rubicon postprocessor produces an Alloy specification equivalent to the verification condition. Each expression type produced by the methods Rubicon stubs corresponds directly to an Alloy expression. Rubicon wraps the verification condition produced above with the appropriate quantifiers, combines it with a set of signatures corresponding to the classes used in the specification, and produces the Alloy specification listed in lines 32-51.

4. EVALUATION

To confirm that Rubicon's analysis is capable of scaling to real applications, we tested it on five open-source Rails applications whose distributions contain RSpec tests written by the original developers. We tried to select applications that perform a variety of tasks and that have a sizable user base. All of the applications we examined contain extensive RSpec test suites with documentation. The five applications we examined are:

- **Insoshi**, a social networking platform
- **Fat Free CRM**, a customer relationship management platform
- **RubyTime**, a time-management system
- **RubyURL**, a URL-shortening service
- **Tracks**, an application to implement the "Getting Things Done" methodology

4.1 Methodology

For each application, we selected a random subset of the files containing developer-written RSpec tests for the application's controllers. The controller tests tend to express properties of the application's behavior, rather than properties about the database schema, so we considered them both a better test of Rubicon's ability to check an application's behavior and more likely to find behavioral bugs in the application.

We converted all of the tests from each selected file into Rubicon specifications by replacing mock objects with quantifiers over objects. This process was straightforward for most tests, since the natural language description of each test combined with the Ruby code implementing it generally described the intent of the test.

We ran both the original RSpec tests and our Rubicon specifications on an Intel Core 2 Duo E7500 with 4GB RAM under Ubuntu 10.04 and Ruby 1.8.7, with the latest version of each application (when available, the development version). Rubicon makes use of version 4.1 of the Alloy Analyzer.

By default, Rubicon uses a finite bound of five during analysis, meaning that the corresponding Alloy Analyzer analysis searches for a counterexample in universes containing five atoms or fewer per Ruby class. We used this bound in our experiments.

Specifications from this evaluation are available on the Rubicon webpage.[2]

4.2 Results

We converted a total of 250 developer-written RSpec tests into Rubicon specifications. The table in Figure 13 contains a summary of the size of each application, the filenames of the test files we converted, the number of tests per file, the average execution time of the original tests and their associated Rubicon specifications, and a comparison between the number of lines of code used before and after the conversion to Rubicon.

The results show that while analyzing Rubicon specifications is several times slower, on average, than executing the original RSpec tests, Rubicon's analysis usually takes only a second or two. Moreover, converting the original RSpec tests into Rubicon specifications made the source code only 11% longer, suggesting that writing Rubicon specifications may not be significantly more difficult than writing RSpec tests.

Figure 14 is a visual representation of the range of per-specification analysis times for each of the test files in Figure 13. For each test file, the bar represents the average analysis time over all of that file's specifications; the extent of the error bars represents the maximum and minimum analysis time for any specification in that file. The maximum analysis time for any test we converted was just over five seconds; more importantly, the maximum and minimum analysis times were always close to the average, suggesting that Rubicon's analysis is consistently fast.

To evaluate the effect of the finite bound on Rubicon's analysis time, we generated the Alloy specifications corresponding to each of the four Rubicon specifications listed in this paper

[2]`http://people.csail.mit.edu/jnear/rubicon`

Filename	Number of Tests	Average RSpec Time per Test	Average Rubicon Time per Test	Original RSpec Lines of Code	Rubicon Lines of Code
Insoshi (12k LOC)					
people_controller_spec.rb	27	0.41s	1.52s	272	314
topics_controller_spec.rb	4	0.52s	1.86s	42	57
comments_controller_spec.rb	14	0.38s	2.08s	126	143
Totals	**45**	**0.44s**	**1.82s**	**440**	**514**
Fat Free CRM (23k LOC)					
home_controller_spec.rb	8	0.64s	2.45s	98	115
comments_controller_spec.rb	21	0.53s	2.12s	254	301
users_controller_spec.rb	27	0.42s	1.87s	343	386
contacts_controller_spec.rb	53	0.44s	1.97s	696	754
Totals	**109**	**0.51s**	**2.10s**	**1391**	**1556**
RubyTime (11k LOC)					
users_spec.rb	31	0.32s	2.11s	271	294
clients_spec.rb	11	0.28s	2.54s	104	132
Totals	**42**	**0.30s**	**2.35s**	**375**	**426**
RubyURL (1k LOC)					
links_controller_spec.rb	8	0.36s	2.63s	73	94
Totals	**8**	**0.36s**	**2.63s**	**73**	**94**
Tracks (22k LOC)					
todo_spec.rb	20	0.27s	2.16s	182	207
user_spec.rb	26	0.23s	2.46s	174	197
Totals	**46**	**0.25s**	**2.31s**	**356**	**404**

Figure 13: Case-Study Summary: the Number of Tests, Average Runtime of Original Test and Corresponding Rubicon Specification, and Lines-of-Code Comparison Between Original Tests and Rubicon Specifications

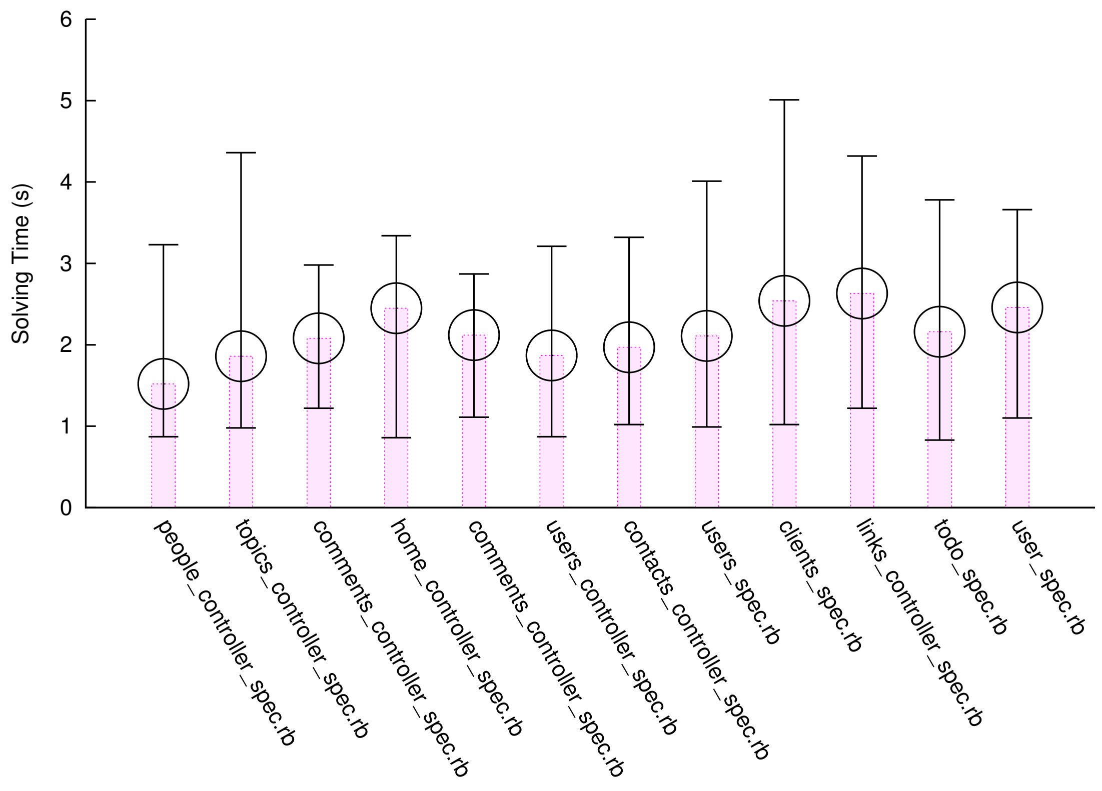

Figure 14: Range of Analysis Times for Rubicon Specifications: Average, Maximum, and Minimum Analysis Times for Each Test File

(in Figures 3, 4, and 6) and manually tested the differences in solving times at different scopes using the Alloy Analyzer. The results of this experiment are displayed in Figure 12. As is common with SAT-based analyses, the solving time begins to increase exponentially as the finite bound rises above ten atoms per class. Rubicon's default bound of five is a compromise attempting to produce consistently fast analysis times without missing bugs.

4.3 Fat Free CRM Bug

Of the RSpec tests we converted, only one led to the discovery of a previously unknown bug. As discussed in Sections 2.2 and 2.3, testing the permissions of both contacts and their associated opportunities, along with the ability of bounded analysis to explore corner cases, allowed us to discover a situation in which a user can view another user's private opportunities.

Several other tests we converted led us to the discovery (also discussed in Section 2.3) that the Fat Free CRM developers had redefined the ActiveRecord `my` method to enforce permissions. Rubicon also redefines this method, and interprets the resulting verification condition as if its use has the semantics originally intended by the Rails developers. The result was a false positive: Rubicon originally reported that any user could view any entity in the system, regardless of permissions. We worked around this problem by renaming the method defined by Fat Free CRM and calling it directly. This was the only Fat Free CRM issue that caused false positives to be reported by Rubicon.

5. RELATED WORK

Rubicon's specification language is based on the RSpec domain-specific language for testing [7]. Our approach of embedding an expressive specification language directly in a programming language is most heavily influenced by QuickCheck [10], a random testing framework for Haskell in which the programmer specifies properties by writing code.

Rubicon's analysis is based on symbolic execution, which was originally developed by King [16] and which has seen both popular use and refinement (e.g. [15]) since its inception. Symbolic execution is itself a form of abstract interpretation, which was formalized by Cousot [11]. The two techniques remain popular with the program analysis community. Rubicon's back-end solver is based on Alloy [14], which is itself based on the Kodkod relational model finder [21].

Existing work on the application of formal methods to web applications focuses on modeling applications, and especially on building navigation models. Bordbar and Anastasakis [4], for example, model a user's interaction with a web application using UML, and perform bounded verification of properties of that interaction by translating the UML model into Alloy using UML2Alloy; other approaches ([17,20]) perform similar tasks but provide less automation.

Most techniques focus specifically on navigation from one page to another, and can analyze only those properties related to possible navigations. Some existing work ([1, 5]) models possible navigations using UML, others ([3]) using directed graphs, and still others ([8, 12, 22]) using statecharts.

Those techniques that allow programmers to specify an application's behavior are closest in their aim to our own work. Syriani and Mansour [19] use SDL (the Specification and Description Language) to model some aspects of application, and provide automatic test case generation based on the model. Haydar et al. [13] use communicating automata to build the application's model, and have explored techniques for verifying properties of those automata. Finally, Andrews et al. [2] use finite state machines to model applications, and also generate test cases. All of these techniques require the programmer to build a separate model of their web application's behavior, and limitations of the modeling technique used mean the set of properties that can be checked is also limited. Rubicon, in contrast, is capable of checking any property expressible in first-order logic.

Techniques that do not require the programmer to build a model of the application tend to focus on the elimination of a certain class of bugs, rather than on full verification. Chlipala's Ur/Web [9] statically verifies user-defined security properties of web applications, and Chaudhuri and Foster [6] verify the absence of some particular security vulnerabilities for Rails applications. Nijjar and Bultan [18] translate Rails data models into Alloy to find inconsistencies.

6. CONCLUSION

Rubicon brings together the ideas of testing, formal specification, symbolic execution, and bounded analysis, and applies them to web applications. Its combination of RSpec and quantifiers is powerful enough to express rich behavioral specifications without requiring programmers to learn a new specification language. Leveraging the Ruby interpreter to perform symbolic execution allows Ruby to execute concrete parts of the application directly, resulting in smaller verification conditions and thus increased scalability. This style of symbolic execution also resulted in decreased development time of the Rubicon tool, since no separate symbolic evaluator had to be developed, and means that users need not install a separate symbolic evaluator. Finally, the relational language and analysis of Alloy and its Analyzer are a perfect match for the database-oriented verification conditions Rubicon generates.

While it remains to be seen if the use of Rubicon is cost-effective, we are encouraged by the ability of our prototype implementation to perform analyses on non-trivial open-source Rails applications, and by the fact that it has already exposed a bug missed in several years of testing. We hope that Rubicon's availability as open-source software will encourage web developers to begin writing and checking specifications.

7. REFERENCES

[1] L. Alfaro. Model checking the world wide web. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 337–349. Springer-Verlag, 2001.

[2] A.A. Andrews, J. Offutt, and R.T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4(3):326–345, 2005.

[3] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *In*

Proceedings of 11th International World Wide Web Conference (WWW'2002). Citeseer, 2002.

[4] B. Bordbar and K. Anastasakis. Mda and analysis of web applications. *Trends in Enterprise Application Architecture*, pages 44–55, 2006.

[5] D. Castelluccia, M. Mongiello, M. Ruta, and R. Totaro. Waver: A model checking-based tool to verify web application design. *Electronic Notes in Theoretical Computer Science*, 157(1):61–76, 2006.

[6] A. Chaudhuri and J.S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 585–594. ACM, 2010.

[7] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North. The rspec book: Behaviour driven development with rspec, cucumber, and friends. *Pragmatic Bookshelf*, 2010.

[8] J. Chen and X. Zhao. Formal models for web navigations with session control and browser cache. *Formal Methods and Software Engineering*, pages 46–60, 2004.

[9] A. Chlipala and LLC Impredicative. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, page 1. USENIX Association, 2010.

[10] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 35(9):268–279, 2000.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[12] L. De Alfaro, T.A. Henzinger, and F.Y.C. Mang. Mcweb: A model-checking tool for web site debugging. In *Poster presented at WWW*, volume 10. Citeseer, 2001.

[13] M. Haydar, A. Petrenko, and H. Sahraoui. Formal verification of web applications modeled by communicating automata. *Formal Techniques for Networked and Distributed Systems–FORTE 2004*, pages 115–132, 2004.

[14] D. Jackson. *Software Abstractions: logic, language, and analysis.* The MIT Press, 2006.

[15] S. Khurshid, C. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, 2003.

[16] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[17] DR Licata and S. Krishnamurthi. Verifying interactive web programs. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 164–173. IEEE.

[18] J. Nijjar and T. Bultan. Analyzing ruby on rails data models using alloy. *GSWC 2010*, page 39, 2010.

[19] J. Syriani and N. Mansour. Modeling web systems using sdl. *Computer and Information Sciences-ISCIS 2003*, pages 1019–1026, 2003.

[20] P. Tonella and F. Ricca. Dynamic model extraction and statistical analysis of web applications. 2002.

[21] E. Torlak and D. Jackson. Kodkod: A relational model finder. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.

[22] M. Winckler and P. Palanque. Statewebcharts: A formal description technique dedicated to navigation modelling of web applications. *Interactive Systems. Design, Specification, and Verification*, pages 279–288, 2003.