Model-Based, Event-Driven Programming Paradigm for Interactive Web Applications

Aleksandar Milicevic Daniel Jackson
Massachusetts Institute of Technology
Cambridge, MA, USA
{aleks,dnj}@csail.mit.edu

Milos Gligoric Darko Marinov
University of Illinois at Urbana-Champaign
Urbana, IL, USA
{gliga,marinov}@illinois.edu

Abstract

Applications are increasingly distributed and event-driven. Advances in web frameworks have made it easier to program standalone servers and their clients, but these applications remain hard to write. A model-based programming paradigm is proposed that allows a programmer to represent a distributed application as if it were a simple sequential program, with atomic actions updating a single, shared global state. A runtime environment executes the program on a collection of clients and servers, automatically handling (and hiding from the programmer) complications such as network communication (including server push), serialization, concurrency and races, persistent storage of data, and queuing and coordination of events.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]: Structured programming; D.2.3 [*Coding Tools and Techniques*]: Object-oriented programming; D.3.2 [*Language Classifications*]: Design languages; D.3.2 [*Language Classifications*]: Very high-level languages; D.2.2 [*Design Tools and Techniques*]: Object-oriented design methods; D.3.4 [*Processors*]: Code generation; I.2.2 [*Automatic Programming*]: Program transformation

General Terms Models, Languages, Events, Software, Design, Web, Frameworks, Security

Keywords model-based; event-driven; distributed; interactive; web applications; declarative programming; automatic programming; software design

Onward! 2013, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509578.2509588

1. Introduction

Today's era of social networks, online real-time user collaboration, and distributed computing brings new demands for application programming. Interactiveness and multi-user experience are essential features of successful and popular applications. However, programming such inherently complex software systems, especially when the interactive (real-time) multi-user component is needed, has not become much easier. Reasons for this complexity are numerous and include:

1. the *distributed architecture* of multiple servers running on the cloud (server farms) interacting with clients running on different platforms (e.g., smartphones, web browsers, desktop widgets, etc.);
2. the *abstraction gap* between the problem-domain level (high-level, often event-driven) and the implementation-level (low-level messages, queues, schedulers, asynchronous callbacks);
3. shared *data consistency*;
4. *concurrency* issues such as data races, atomicity violations, deadlocks, etc.

Problems of this kind are known as *accidental complexity* [13], since they arise purely from abstraction mismatches and are not essential to the actual problem being solved. Carefully managing accidental complexity, however, is absolutely crucial to developing a correct and robust system. Although thoroughly studied in the literature, these problems not only pose serious challenges even for experienced programmers, but also distract the programmer from focusing on essential problems, i.e., designing and developing the system to achieve its main goals.

We propose a new model-based *programming paradigm* for designing and developing interactive event-driven systems, accompanied by a *runtime environment* for monitored execution of programs written in that language. Our paradigm is structured around models (mostly declarative, but fully executable) using concepts from the domain of interactive web applications, (e.g., shared data, system events, interactions and interconnections between clients, etc.), and

also explicitly separating concerns like data, core business logic, user interface, privacy and security rules, etc. This allows the programmer to think and write code at a high-level, close to the actual problem domain, directly addressing the abstraction gap issue.

The structural information about the system, which is inherently present in these models, allows the runtime environment to automatically manage many forms of accidental complexity, from synchronizing and dispatching concurrent events to propagating data updates to all connected clients (also known as "server push" in the web developers community). The programmer, therefore, has a very simple sequential programming view, and it is the job of the runtime environment to turn that into a distributed application. Relieving the programmer of writing multithreaded code eliminates, by construction, a whole class of concurrency bugs, which are notoriously difficult to debug and fix.

We call this whole approach SUNNY, as our goal is to shine some light on the dark world of distributed systems, making it less tedious and more fun, and, at the same time, more robust and more secure. In this paper, we also present a concrete implementation of this approach for Ruby on Rails, which we call RED (Ruby Event Driven).

2. Example

In this section we present a simple example of a real-world application to explain the proposed programming paradigm and illustrate the expressiveness and ease of use of our language.

Our intention in this example is to implement a "public IRC" (Internet Relay Chat) web application, meaning that anyone can create a chat room (provided that a room with the same name does not already exist) and that the existing rooms are public (anyone can join and send messages once joined). With most applications of this kind, the web GUI must be responsive and interactive, automatically refreshing parts of the screen whenever something important happens (e.g., a new message is received), without reloading the whole page.

Figure 1 shows a simple IRC implementation written in RED (our implementation of SUNNY for Ruby on Rails). RED programs consist of several different *models* of the system (described next), and as such are fully executable. These models are fairly high-level and mostly declarative, so we occasionally refer to them as *specifications*.

The **data model** of the IRC application consists of a `User` record (which specializes the RED library `AuthUser` record and adds a status field), a `Msg` record (where each message has a textual body and a sender), and a `ChatRoom` record (each room has a name, a set of participating users, and a sequence of messages that have been sent). These fields are defined using the `refs` and `owns` keywords: the former denotes aggregation (simple referencing, without any constraints), and the latter denotes composition (implying that (1) when a record is deleted, all owned records should be deleted, and (2) no two distinct records can point to the same record via the same owned field).

The **network model** in this example consists of two machines, namely `Server` and `Client`. The `Client` machine has a corresponding `User`, whereas the `Server` machine maintains a set of active `ChatRoom`s. They respectively inherit from the library `AuthClient` and `AuthServer` machines, to bring in some fairly standard (but library-defined, as opposed to built-in) user management behavior, like new user registration, sign-in and sign-out events, etc.[1]

To implement the basic functionality of IRC, we defined an **event model** with three event types: `CreateRoom`, `JoinRoom`, and `SendMsg`, as shown in Figure 1(c).

Each event has an appropriate precondition (given in the `requires` clause) that checks that the requirements for the event are all satisfied before the event may be executed. For instance, events `CreateRoom`, `JoinRoom`, and `SendMsg` all require that the user has signed in (`client.user` is non-empty), `SendMsg` requires that the user has joined the room, etc.

A specification of the effects of an event (given in the `ensures` clause) is concerned only with updating relevant data records and machines to reflect the occurrence of that event. For example, the effects of the `JoinRoom` event amount to simply adding the user requesting to join the room to the set of room members; the runtime system will make sure that this update is automatically pushed to all clients currently viewing that room. Actions like updating the GUI are specified elsewhere, independently of the event model; this is a key to achieving separation of concerns.

By default, all fields in our models are public and visible to all machines in the system. That approach might be appropriate for the running "public IRC" example, where everything is supposed to be public anyway. For many other systems, however, it is often necessary to restrict access to sensitive data. Let us therefore define some privacy rules even for this example to show how that can be done in SUNNY, declaratively and independently of the event model.

The `HideUserPrivateData` policy from Figure 1(d) dictates that the value of a user's password should not be revealed to any other user and, similarly, that the status message of a user should not be revealed to any other user, unless the two users are currently both members of the same chat room. Note that the latter rule is dynamic, i.e., it depends on the current state of the system (two users being together in a same chat room) and thus its evaluation for two given users may change over time.

In addition to restricting access to a field entirely, when a field is of a collection type, a policy can also specify a filtering condition to be used to remove certain elements from

[1] The full listing of the `RedLib::Web::Auth` library is given in Figure 2; several events defined in this library are referred to later in the text.

(a) data model

```
record User < AuthUser do
  # inherited fields
  #   name: String,
  #   email: String,
  #   pswd_hash: String,
  refs status: String
end
```

```
record Msg do
  refs text: Text,
       sender: User
end
```

```
record ChatRoom do
  refs name: String,
       members: (set User)
  owns messages: (seq Msg)
end
```

(b) network model

```
machine Client < AuthClient do
  refs user: User
end

machine Server < AuthServer do
  owns rooms: (set ChatRoom)
end
```

(c) event model

```
event CreateRoom do
  from client: Client
  to   serv: Server

  params roomName: String

  requires {
    client.user    &&
    roomName       &&
    roomName != "" &&
    !serv.rooms.find_by_name(roomName)
  }

  ensures {
    room = ChatRoom.create
    room.name = roomName
    room.members = [client.user]
    serv.rooms << room
  }
end
```

```
event JoinRoom do
  from client: Client
  to   serv: Server

  params room: ChatRoom

  requires {
    u = client.user
    client.user &&
    !room.members.include?(u)
  }

  ensures {
    room.members << client.user
  }
end
```

```
event SendMsg do
  from client: Client
  to   serv: Server

  params room: ChatRoom,
         msgText: String

  requires {
    client.user &&
    room.members.include?(client.user)
  }

  ensures {
    msg = Msg.create
    msg.text = msgText
    msg.sender = client.user
    room.messages << msg
  }
end
```

(d) security model

```
policy HideUserPrivateData do
  principal client: Client

  # restrict access to passwords except for owning user
  restrict User.pswd_hash.unless do |user|
    client.user == user
  end

  # restrict access to status messages to users
  # who share at least one chat room
  # with the owner of that status message
  restrict User.status.when do |user|
    client.user != user &&
    ChatRoom.none? { |room|
      room.members.include?(client.user) &&
      room.members.include?(user)
    }
  end
end
```

```
policy FilterChatRoomMembers do
  principal client: Client

  # filter out anonymous users (those who have not
  # sent anything) from the 'members' field
  restrict ChatRoom.members.reject do |room, member|
    !room.messages.sender.include?(member) &&
    client.user != member
  end
end
```

Figure 1. A full implementation (excluding any GUI) of a simple public IRC application written in RED, our new domain-specific language for programming event-driven systems. Since RED is very high-level and mostly declarative, we often refer to RED programs as *models*, and also consider them to be the *specification* of the system.

that collection before the collection is sent to another machine. The `FilterChatRoomMembers` policy hides those members of a chat room who have not sent any messages (this simulates, to some extent, "invisible users", a feature supported by some chat clients).

SUNNY automatically checks policies at every field access; if any policy is violated the access is forbidden simply by replacing the field value with an empty value.

3. Why The World Needs SUNNY

Interactive multi-user applications, even when having relatively simple functional requirements, are difficult to write using today's programming languages and available state-of-the-art frameworks, the main reason being the abstraction gap between the problem domain and the concepts available at the implementation level.

Just as one example, current systems typically do not offer much help with structuring and organizing the system

```
def hash_pswd(str)
  Digest::SHA256.hexdigest(pswd_plain)
end

abstract_record AuthUser, {
  name: String,
  email: String,
  pswd_hash: String
} do
  def authenticate(pswd_plain)
    pswd_hash == hash_pswd(pswd_plain)
  end
end

abstract_machine AuthServer {}
abstract_machine AuthClient { user: AuthUser }

event Register do
  from client: AuthClient
  params name: String, email: String, pswd: String
  requires { !AuthUser.find_by_email(email) }
  ensures  {
    client.create_user! :name => name,
                        :email => email,
                        :pswd_hash => hash_pswd(pswd)
  }
end

event SignIn do
  from client: AuthClient
  params email: String, pswd: String
  ensures {
    u = AuthUser.find_by_email(email)
    fail "User #{email} not found" unless u
    pswd_ok = u.authenticate(pswd)
    fail "Wrong password for #{email}" unless pswd_ok
    client.user = u
  }
end

event SignOut do
  from client: AuthClient
  requires { some client.user }
  ensures  { client.user = nil }
end

event Unregister do
  from client: AuthClient
  requires { client.user }
  ensures  { client.user.destroy }
end
```

Figure 2. The `RedLib::Web::Auth` library module, written in RED itself, provides common records and events for managing users and user authentication.

around events, despite proper event handling being at the core of most interactive applications. Instead, they offer callbacks, which can be registered from any source code location, almost inevitably leading to what is known as *callback hell* [20]. As a consequence, programs end up being cluttered, the flow structure becomes very difficult to infer from the source code, leading to programs that are hard to understand and maintain.

Other than event-handling, the programmer has to face a number of other technological barriers, including concurrency, object-relational mapping, server push, etc. Even though these technological barriers have been individually overcome, the solutions sometimes come in a form of best-practices or guidelines, so the programmer still has to spend time implementing them for the project at hand, which is, for the barriers mentioned above, time-consuming, tedious, and also error-prone.

We illustrate these points in terms of three concrete platforms for developing web applications.

3.1 The Java Approach

The Java language, which gained much of its success from being proposed as a platform for web development, is still one of the top choices for development of enterprise web systems. The language being mature, the runtime (JVM) being fast and solid, and an abundance of freely available third-party libraries are some of the points in favor.

The trend of web development in Java still seems to be based around manually configuring and integrating a multitude of standalone, highly specialized libraries, designed independently to solve various web-related tasks, as opposed to having a single overarching framework designed to address most of the common issues. A highly experienced Java expert, who is already familiar with the existing libraries for web development, object-relational mapping, database management, server push, and such (also already knowing how to configure them all so that they can interoperate and work well together) would have a good start developing our IRC example. For the rest of us, however, the situation is much worse. For someone already familiar with Java, but not too familiar with web development in Java, the effort just to get a handle on all the necessary libraries would by far exceed the effort needed to implement the functionality of our example.

Even the expert would have to be very careful about managing concurrent requests on the server side, setting up event processing queues (to avoid common concurrency issues but still achieve good throughput), implementing corresponding producer and consumer threads, and so on. Probably equally cumbersome would be manually keeping track of which clients are viewing what, automatically propagating updates when the data underlying the views change, and implementing Ajax-style code on the client side to refresh the GUI smoothly. All these are generic enough tasks, for which the implementation does not seem to differ much from one project to another, so it seems unfortunate that they have to be repeated every time. One of the design goals of SUNNY was to explicitly address this issue, and let the framework, not the programmer, fight the technology.

3.2 The Rails Approach

In contrast to Java, the design of Rails [6] adopted the "convention over configuration" school of thought: instead of manually configuring every single aspect of the application, if certain conventions are followed, the Rails framework will automatically perform most of the boilerplate tasks behind the scene and "magically" make things happen.

Underneath the surface, unfortunately, it is still a configuration mess, and the magic is mostly concerned with low-level configuration of different components and how to tie

them all together. This creates problems for many Rails programmers, because, as this magic has no high-level semantics, it is often difficult to understand and remember not only how it works, but also what it does. In SUNNY, we aim to offer a different kind of magic, which is easy to understand at the conceptual level (e.g., data updates are automatically propagated to clients, all the way to automatically refreshing the GUI), so the programmer need not understand the technical details behind its implementation.

By imposing some structure on how the system should be organized and implemented (e.g., using the Model View Controller (MVC) architecture), Rails can indeed provide a lot of benefits for free. One of the most appealing features of Rails (especially back when it first appeared) is "scaffolding": given just a few model files describing how the data structures are organized, Rails automatically generates a running web application, with the full stack, from the database to the web server, automatically configured and set up.

While scaffolding greatly reduces the startup cost of developing a new application (even for inexperienced programmers), it is not meant to be a permanent, system-level solution. The reason is that it is based on code generation from *transient* models: the generated files (including database configuration files, Rails controller classes, HTML views) work fine at the beginning, but as soon as something needs to be changed, everything needs to be changed manually, since there is nothing to keep them in sync otherwise. Furthermore, the models used for scaffolding support only scalar, primitive-typed fields. In SUNNY, in contrast, models (like those shown in Figure 1) are first-class citizens; not only do they exist at runtime, but they are central to the whole paradigm (i.e., the entire runtime semantics is built around them). Our models are also much richer, so there is enough information available to the SUNNY runtime environment to interpret them on the fly, instead of generating code up front. That way, the common problem of having inconsistencies between the models and the code is eliminated in SUNNY.

Concurrency in Ruby is an interesting topic. Ruby is inherently not concurrent (because of a Global Interpreter Lock). As a result, Rails programmers can safely ignore threads and synchronization, and still have no data race issues. This, of course, comes at the cost of low scalability. When a more scalable implementation is needed, typically solutions require that the system is restructured so that blocking operations (like I/O) are offloaded to a different process, which is at the same time told what to do upon completion of the requested operation (the so called *Reactor* pattern). Refactoring a system in this manner is almost never trivial nor straightforward.

We believe that concurrency and parallel processing do not have to be sacrificed to this extent to give the programmer a safe sequential programming model, as explained in more detail in Section 4.1.

3.3 The Meteor Approach

Meteor [5] is a newer web framework for fast and convenient development of modern web applications. Meteor has been rapidly gaining popularity. It is a pure JavaScript implementation (both server and client have to be written in JavaScript) of an event-driven (publish/subscribe) system which also automatically propagates updates to all connected clients whenever the shared data changes.

Unlike SUNNY, Meteor focuses on providing a platform for automatic data propagation, whereas SUNNY is designed to also handle other aspects of the system, including richer models for shared data, GUI scaffolding, automated support for concurrency, etc. Specifically, Meteor does not offer much structure to help design the system, nor does it have rich models of the underlying shared data. The data model in Meteor consists of a number of flat collections (corresponding directly to database tables), with no type information, and no explicit relationship between different model classes. Rich models enable both software engineering benefits (like automated test generation and verification of end-to-end properties), as well as productivity benefits (like automated GUI scaffolding)[2].

4. The SUNNY Approach

A key idea of SUNNY is to make it possible to think about different events in isolation, and only in terms of modifications to the data model they entail. Therefore, in the design phase, the programmer does not have to think about other issues, such as how to update the user interface to reflect the changes, or even about security and privacy policies; those can be specified separately and independently from the core event model. Limiting the specification this way is what forces the programmer to focus on the core logic of the system first (hence reducing the chances of software bugs in those core parts of the system) and what enables us to provide a unified and overarching runtime environment for fully automated resource management and constant data access monitoring for security violations.

The main components of SUNNY are:

- a Domain Specific Programming Language
- a Runtime Environment
- an Online Code Generator
- a Dynamic Template-Based Rendering Engine

Instead of going into technical details about each of these components, our main intent for this paper is to provide a broader discussion of the big-picture goals and design behind SUNNY, illustrate its usefulness and practicality through examples, and argue for the benefits it brings to

[2] In contrast to GUI scaffolding implemented in Rails, ours is not a one-off code generation approach; it is rather based on generic (application-agnostic) templates which get evaluated at runtime, so again, there is no problem of falling out of sync.

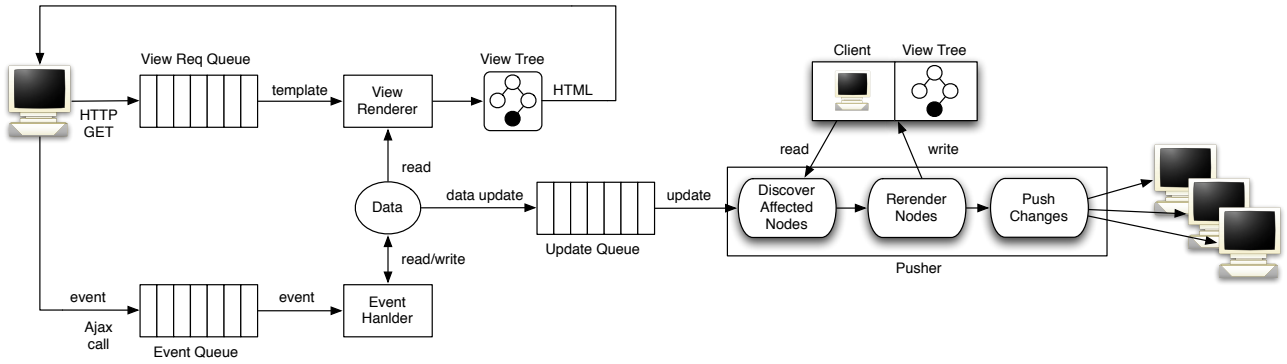

Figure 3. Internal architecture of SUNNY's runtime environment for concurrent processing of events and user requests.

software engineering best practices. We will next, therefore, walk through a sample execution of our system (still using the running IRC example) to better illustrate how the system works and how the benefits are achieved. Afterward, we will briefly describe each of the mentioned components using the concrete syntax of RED.

4.1 Sample Execution

Consider a scenario in which a user initially opens the home page of our IRC application. This request is received by the web server via the HTTP GET method and placed in a processing queue (namely *View Req Queue*, Figure 3, top pipeline). From there, it is picked up by the *View Renderer* component, while the web server can immediately go back to serving incoming requests.

Let us assume that the view corresponding to the home page is the *irc* template shown in Figure 4(a) and that the user is not logged in yet. These templates are written in the ERB language, which allows arbitrary Ruby expressions to be embedded inside the `<% %>` and `<%= %>` marks (the difference being that only the latter produces a textual output, while the output of the former is ignored). The *View Renderer*, therefore, evaluates the "else" branch of the template, and returns a login page with two input fields (for email and password) and a "Sign-in" button.

While rendering a template, the *View Renderer* also maintains a *View Tree* structure which holds a single node for each Ruby expression that was evaluated during the execution of the template (templates can invoke other templates, potentially creating a hierarchy of nodes). Each node stores a list of fields that were accessed while the corresponding expression was being evaluated. In the case of this example, there is only one node in that tree, and the only field that was accessed was the `user` field of the current client instance (during the evaluation of the "if" condition).

On the client side, our JavaScript library automatically recognizes the "Sign-in" button by its `data-trigger-event` HTML5 attribute, and, according to its value, associates it with the `SignIn` event (which is a part of the previously imported `Auth` library (Figure 2)). More concretely, it assigns an "onclick" handler to it, so that when the button is clicked, the associated form (discovered via the `data-params-form` attribute) is submitted (via an Ajax call) as the parameters of the `SignIn` event.

When the user clicks this button, the `SignIn` event is triggered and received on the server side via the bottom processing pipeline in Figure 3. The *EventHandler* then picks it up from the queue, checks its precondition, and if it holds (in this case it does, since the `requires` method is empty), proceeds to execute its postcondition. Assuming that the user entered valid credentials, the execution will assign value to the `user` field of the current `client` instance (the `client` instance is always implicit and denotes the machine which submitted the currently executing event).

Any modification to the data model triggers an internal "data update" signal, which is placed in the *Update Queue* (the right-most pipeline in Figure 3). A component called *Pusher* is in charge of serving the *Update Queue*. Every time an update is received, it goes through a list of all connected clients and corresponding view trees, discovers which nodes could potentially be affected by the current update (by checking their list of field accesses), re-renders those nodes, updates the global *Client* $\rightarrow$ *View Tree* map, and pushes those changes to clients. On the client side, only the corresponding portion of the HTML DOM is replaced by the newly rendered text.

In the running scenario, the only node that was stored for the current client was dependent on the `user` field, so only it has to be re-rendered. The new content is produced by executing the "then" branch, which amounts to rendering the *user.html.erb* template for the current user (the user object is by default available to the callee template via the `user` variable), and rendering the *chat_room.html.erb* template once for each room on the server (in this case the default variable name would be "chat_room", but it is instead explicitly set to "room" via the `:as` option).

The execution then continues in the same manner: clients continue to perform actions by triggering events from the

domain, and the server keeps processing events, detecting changes in the data model, and re-rendering parts of the client views when needed. An explanation of how asynchronous message sending is declaratively specified directly in an HTML template (no separate JavaScript file), and without any Ajax code, is given in Section 4.4.3.

To get a running version of this sample execution, if using RED, the programmer only needs to:

- write the data, machine, and event models from Figure 1 (the security model is not necessary);
- write the HTML templates from Figure 4;
- deploy the application to a server running Ruby on Rails with our extensions; and
- set the application home page to *irc.html.erb* (by configuring the root route in Rails).

Comparing to implementing the same application in standard Rails:

- in place of our data model, the programmer would write ActiveRecord model classes (one model class per record), which are more verbose and require more configuration (as discussed in Section 4.4.2);
- in place of our machine model, the programmer would likely use in-memory classes and the Rails session storage (not affecting the complexity of the implementation);
- in place of our event model, the programmer would write controllers of approximately the same complexity;
- the HTML templates would remain the same, as well as the deployment process.

Additionally, the Rails programmer would have to

- write a database schema (discussed in Section 4.4.1), carefully following the Rails naming convention;
- write a controller for each model class implementing the standard CRUD (Create, Read, Update, Delete) operations (again, certain naming conventions have to be followed);
- configure routes for each controller;
- decide on a third party library to use to implement the server push (pushing data updates to connected clients in real time);
- implement server-side code that keeps track of what data each client is currently displaying;
- implement server-side code that detects model changes (made during the execution of controllers);
- implement server-side code that pushes data changes to each client whenever a piece of data currently being displayed on that client is changed;
- implement client-side code that listens for data changes from the server;
- implement client-side code that dynamically re-renders affected parts of the GUI whenever a data update is received.

In both cases, a CSS file is necessary in order to make the GUI look pretty.

While RED provides dynamic GUI updates for free, and for that does not require the programmer to write any JavaScript, it does not prevent him or her from doing so; RED comes with a client-side JavaScript library (see Section 4.4.3) which can be used to interact with the server-side, customize how the GUI gets updated (e.g., implement special visual effects or animations), asynchronously trigger events, etc.

4.2 Domain-Specific Programming Language

We designed a domain-specific language for writing SUNNY models in order to better emphasize the key concepts of our paradigm. This language has strong foundations in the Alloy modeling language [33], a *relational* language based on first-order logic. Alloy is declarative, and as such, it is not executable per se; it is instead used primarily for modeling and checking logical properties of various kinds of systems. Most of Alloy's expressive power comes from its relational base (including all the supported relational operators), which, however, can be efficiently executable in the context of object-oriented programs [57]. For example, the dot operator ('`.`') is actually a relational join, so an expression that fetches all users currently present in any chat room on a server can be written simply as `Server.rooms.members`.

In RED, we implemented this language as an embedded DSL in Ruby. Concretely, each of `record`, `machine`, and `event` is just a function that takes a name, a hash of field $name \rightarrow type$ pairs, and a block; it (1) returns a `Class` having those field names as attributes, while storing the type information in a separate meta-class, (2) creates, in the current module, a constant with the given name and assigns the newly created class to it, and (3) if a block is given, evaluates that block in the context of that class. The block parameter can be used to define additional instance methods (as in Figure 2, method `authenticate` in record `AuthUser`), but also to define fields with more information than just name and type (e.g., the call to the `owns` function in Figure 1(a), which additionally specifies that whenever a chat room is deleted, the deletion operation should be propagated to all the messages referenced by that room via the `messages` field).

Having this syntactic sugar (instead of just using the built-in `class` keyword) provides a convenient way of specifying typed fields, but more importantly, being in charge of class generation also gives us an easy way to hook into all field accesses, where we perform the necessary policy checks. We override the `const_missing` method, so that unresolved types are converted to symbols at declaration time; we only require that all types can be resolved at runtime.

(a) template file: *irc.html.erb*

```
<% if client.user %>
 <%= render client.user %>
 <%= render server.rooms, :as => 'room' %>
<% else %>
 <form id="login-form">
  Email:    <input type="text" name="email"/>
  Password: <input type="password" name="password"/>
 </form>
 <button data-trigger-event="SignIn"
         data-params-form="login-form">
    Sign In</button>
<% end %>
```

(b) template file: *user.html.erb*

```
<div class="User">
 Welcome <%= user.name %> (<%= user.email %>)
</div>
```

(c) template file: *chat_room.html.erb*

```
<div class="ChatRoom">
 Name:    <%= room.name %>
 Members: <%= room.members.name.join(", ") %>
 Posts:   <%= render room.messages %>


 <input id="txt-<%=room.id%>" type="text"/>
 <button data-trigger-event="SendMsg"
         data-param-room="${new ChatRoom(<%=room.id%>)}"
         data-param-msgText="${$('#txt-<%=room.id%>').val()}">
   Send</button>
</div>
```

(d) template file: *msg.html.erb*

```
<div class="post">
 <%= msg.sender.name %>: <%= msg.text %>
</div>
```

Figure 4. HTML views for the IRC example from Figure 1 written using ERB templates (the ERB language allows arbitrary Ruby code to be embedded and evaluated inside the `<% %>` and `<%= %>` marks).

Note that, however, none of our language features mandated this implementation choice; a different implementation targeting a different platform is possible.

4.3 Runtime Environment

One of our main goals is to relieve the programmer of having to explicitly implement a distributed system, i.e., explicitly synchronize multiple processes, handle inter-process communication, manage queues and messages, ensure data consistency, and a number of other tasks typical for distributed and concurrent programming. By introducing a specific programming model (as described previously), we tailored a generic runtime environment to automate all those tasks. The runtime implements a standard message-passing architecture, a well-known and widely used idiom for designing distributed systems, which we use to dispatch events and data updates between entities (Figure 3).

Another important role of the runtime environment is to automatically check and enforce privacy policies at runtime. Policies are checked at every field access attempted by the user-defined code: all relevant restriction rules are discovered and applied. Instead of throwing an exception when the access is forbidden, an empty relation is returned. This makes it possible for the client code to be written in a mostly policy-agnostic style. For example, the client code can simply say `room.members` to fetch the members of a chat room, and rely on the runtime to return only those elements that the client is allowed to see.

Policies are also considered when objects are being serialized (by the runtime) prior to being sent to another machine (e.g., as part of the automatic propagation of data updates). Consider a client (`client`) attempting to fetch all rooms by executing `server.rooms`. This is a legal operation, as all field accesses are permitted (it is a public IRC application). Any sensitive data (e.g., the password and status fields of the room members), however, must still be hidden or their values properly filtered out, which our runtime does automatically (simply by returning an empty relation every time access is forbidden).

This illustrates the declarative nature of our privacy policies, and how the runtime can automatically enforce them. It also shows that the operational code (e.g., event handlers, embedded GUI formulas, etc.) usually can be written independently of privacy policies, and does not need to be updated when policies change.

4.4 Online Code Generator

Many of the responsibilities of the runtime environment are enabled by the code automatically generated from the core models, on the fly, during the system initialization phase. In addition, we use code generation to automate various common tasks. Several of these tasks are briefly described next.

4.4.1 Database Migrations

The richness of our data model makes it possible for us to handle data persistence fully automatically. This includes (1) generating and maintaining a database schema (discussed in this section), and (2) implementing an object-relational mapper (i.e., mapping domain objects onto that schema, discussed next in Section 4.4.2).

A database schema provides a way to persist all relevant information from the domain model. Because the schema is always supposed to closely mirror the model, ideally it should not have to be written/programmed separately. In standard Rails, however, that is not the case; the schema exists as a standalone code artifact, and the programmer is in charge of maintaining it and keeping it in sync with the application model. Although Rails comes with automated generators that can create schema skeleton files from sim-

(a) auto-generated Rails migration file

```
class UpdateTables < ActiveRecord::Migration

 def change
  create_table :auth_clients do |t|
    t.column :auth_token, :string
    t.references :user
    t.column :user_type, :string
    t.references :user
    t.column :user_type, :string
    t.column :type, :string
    t.timestamps
  end

  create_table :auth_servers do |t|
    t.column :type, :string
    t.timestamps
  end

  create_table :auth_users do |t|
    t.column :name, :string
    t.column :email, :string
    t.column :password_hash, :string
    t.column :status, :string
    t.column :type, :string
    t.timestamps
  end

  create_table :msgs do |t|
    t.column :text, :text
    t.references :sender
    t.column :sender_type, :string
    t.references :chat_room_as_message
    t.timestamps
  end

  create_table :chat_rooms do |t|
    t.column :name, :string
    t.references :server_as_room
    t.column :server_as_room_type, :string
    t.timestamps
  end

  create_table :chat_rooms_users_members,
               :id => false do |t|
    t.column :chat_room_id, :int
    t.column :user_id, :int
  end

  create_table :sessions do |t|
    t.string :session_id, :null => false
    t.text :data
    t.timestamps
  end
  add_index :sessions, :session_id
  add_index :sessions, :updated_at

 end

end
```

(b) auto-generated ActiveRecord classes

```
class Msg < Red::Model::Record # < ActiveRecord::Base
 attr_accessible :text
 belongs_to      :sender,
                 :class_name  => "User",
                 :foreign_key => :sender_id
 belongs_to      :chat_room_as_message,
                 :class_name  => "ChatRoom",
                 :foreign_key => :chat_room_as_message_id,
                 :inverse_of  => :messages

 # interceptors for field getters and setters
 def text()      intercept_read(:text)      { super } end
 def text=(val)  intercept_write(:text, val){ super } end
 ...
end

class ChatRoom < Red::Model::Record # < ActiveRecord::Base
 attr_accessible         :name
 has_and_belongs_to_many :members,
                         :class_name              => "User",
                         :foreign_key             => :chat_room_id,
                         :association_foreign_key => :user_id,
                         :join_table  => "chat_rooms_users_members"
 has_many                :messages,
                         :class_name  => "Msg",
                         :foreign_key => :chat_room_as_message_id,
                         :dependent   => :destroy
 belongs_to              :server_as_room,
                         :class_name  => "Server",
                         :foreign_key => :server_as_room_id,
                         :inverse_of  => :rooms

 # interceptors for field getters and setters
 def name()      intercept_read(:name)      { super } end
 def name=(val)  intercept_write(:name, val){ super } end
 ...
end

class User < RedLib::Web::Auth::AuthUser # < Red::Model::Record
  attr_accessible         :status
  has_and_belongs_to_many :chat_rooms_as_member,
                          :class_name              => "ChatRoom",
                          :foreign_key             => :user_id,
                          :association_foreign_key => :chat_room_id,
                          :join_table => "chat_rooms_users_members"
  has_many                :msgs_as_sender,
                          :class_name  => "Msg",
                          :foreign_key => :sender_id,
                          :inverse_of  => :sender
  has_many                :clients_as_user,
                          :class_name  => "Client",
                          :foreign_key => :user_id,
                          :inverse_of  => :user

 # interceptors for field getters and setters
 def status()      intercept_read(:status)      { super } end
 def status=(val)  intercept_write(:status, val){ super } end
 ...
end
```

Figure 5. Several different snippets of automatically generated code for the IRC example: (a) full database migration file (in Ruby), creating a schema for the persistent entities from the domain (records and machines), (b) excerpt from the translation of domain records to `ActiveRecord` classes, with mappings of fields to database columns and field interceptors.

ple $name \rightarrow type$ pairs, they only work for primitive types and scalar fields; for more advanced features like type inheritance and non-scalar fields (many-to-many relations), the programmer has to manually extend the generated schema file in such a way that it works with the object-relational mapper on the other side.

Figure 5(a) gives a full listing of the database schema (in the form of a Ruby migration class, standard for the Rails framework) that RED automatically generated for the IRC example. This schema supports all the features of the model, so the programmer does not even have to look at it.

ActiveRecord (the object-relational mapper used in Rails and in our framework) implements the *single table inheri-*

tance strategy for handling inheritance. Hence, for each base type we generate one table with columns for all fields of all of its subtypes, plus an extra string-valued column (named `:type`) where the actual record type is stored. For example, in the `:auth_users` table, the first three columns correspond to fields from `AuthUser` and the fourth column is for the single field from `User`. Furthermore, in every other table referencing such a table, an additional "column type" column must be added to denote the declared type of the corresponding field, as in the `:msg` table (columns `:sender` and `:sender_type`).

When a record field type is of arity greater than 1, a separate join table must be created to hold that relation. This is the case with the `ChatRoom.members` field (referencing a set of `User`s). The corresponding join table (`:chat_rooms_users_members`) stores all tuples of the `:members` relation by having each row point to a row in the `:chat_rooms` table and a row in the `:users` table. In the special case when a field owns a set of records (e.g., field `ChatRoom.messages`, meaning that a given message can be referenced via that field by at most one chat room), instead of a join table, a referencing column is placed in the table corresponding to the type of that field (the `:chat_room_as_message` column in table `:msgs`).

The last `create_table` statement in Figure 5(a) simply creates a table where the session data will be stored, and is independent of the domain data model.

Despite being mostly straightforward, writing migrations by hand is still tedious and time consuming, and, for developers new to Rails, can often be a source of mysterious runtime errors. Even after those initial errors have been fixed, the gap between the schema and the application model still remains. RED eliminates all these issues by having a single unified model of the system and automatically driving various implementation-level technologies (such as the database schema maintenance) directly from it.

4.4.2 ActiveRecord Classes and Reflections

As we explained in Section 4.2, the `record` keyword in RED is actually implemented as a Ruby function that creates a new class and assigns a named constant to it. Here we discuss the generated record classes (listed in Figure 5(b)) in more detail.

ActiveRecord provides "*reflections*" for specifying associations between *models* (i.e., records in our terminology). Primitive fields are declared with `attr_accessible` (e.g., `ChatRoom.name`), one-to-many associations with `has_many` on one side and `belongs_to` on the other (e.g., `ChatRoom.messages`), and many-to-many associations with `has_and_belongs_to_many` (e.g., `ChatRoom.members`). Various options can be provided to specify the exact mapping onto the underlying database schema.

As with migration generators, Rails provides generators for ActiveRecord model classes as well, but again, with limited features and capabilities. While most of the schema-mapping options (e.g., `:foreign_key`, `:join_table`, `:association_foreign_key`) can be omitted if the naming convention is followed when the schema is written, the programmer still has to manually write these reflections for all but primitive fields. Furthermore, ActiveRecord requires that reflections are written on both sides of an association, meaning that each non-primitive field has to have its inverse explicitly declared in the opposite class (which is another step that our system eliminates). Finally, even though the database schema and the model classes are coupled, there is nothing that keeps them in sync in standard Rails. This not only makes the development process more cumbersome and error-prone, but also makes it difficult to perform any system redesign or refactoring.

Controlling the generation of model classes also lets us intercept all field accesses, where we perform all the necessary security checks, detect changes to the data model for the purpose of updating client views, wrap the results of getter methods to enable special syntax (e.g., the Alloy-style relational join chains), etc.

4.4.3 JavaScript Model for the Client-Side

One of the main ideas behind SUNNY is to have a single unified model of the system, and a model-based programming paradigm that extends beyond language and system boundaries. In RED, we wanted to preserve this idea and enable the same kind of model-based programming style on both the server side and the client side, despite the language mismatch. More concretely, we wanted to provide the same high-level programming constructs for instantiating and asynchronously firing events in JavaScript on the client side, as well as constructing and manipulating records.

To that end, we translate the system domain model into JavaScript, to make all the model meta-data available on the client side. We also implemented a separate JavaScript library that provides prototypes for the generated model classes, as well as many utility operations.

Figure 6 gives an excerpt from the translation of the IRC application's domain model. Up on the top are constructor functions for all records, machines, and events. The `mk_record` and `mk_event` functions (part of our library) take a name and (optionally) a super constructor, and return a constructor function with the given name and a prototype extending the super constructor's prototype. This is followed by the meta-data for each record, machine, and event, which contains various information about the type hierarchy, fields, field types, etc. All this information is necessary for our library to be able to provide generic and application-agnostic operations. One such operation we mentioned before, in Section 4.1, where we talked about how DOM elements having the `data-trigger-event` attribute are automatically turned into event triggers.

Let us finally take a look at how asynchronous message sending is implemented on the client side, that is, how such

an operation can be specified declaratively, directly in an HTML template file, without writing any Ajax code.

The *chat_room.html.erb* template (Figure 4(c)) contains a text input field and a send button, with the intention to trigger the `SendMsg` event and send whatever message is in the text input field whenever the send button is pressed. To achieve that, we added three HTML5 data attributes to the send button element; we used `data-trigger-event`, as before, to denote the type of the event, and two `data-param` attributes to specify the two mandatory arguments of the `SendMsg` event, `room` and `msgText`.

The value for the `room` parameter is known statically—it is exactly the chat room object for which the *chat_room* template is being executed. However, that value is an object, so it is not possible to directly embed it in the template as a string-valued attribute. Instead, we inline a small piece of JavaScript code that, when executed, creates an equivalent room on the client side. Knowing the id of that room, and having a full replica of the model classes on the client, that code is as simple as `new ChatRoom(<%=room.id%>)`[3]; we only need to tell our JavaScript library that the value we are passing is not a string, but code, by enclosing it in `${}`[4].

The value for the `msgText` parameter is not known statically, and has to be retrieved dynamically when the user presses the send button. As in the previous case, we can specify that by inlining a piece of JavaScript that finds the input text field by its id (using the jQuery syntax `$('#<id>')`) and reads its current value (by calling the `.val()` function).

An alternative approach to declaratively specifying event parameter bindings, that would require no JavaScript from the programmer, would be to somehow annotate the input text field (e.g., again by using the HTML5 data attributes) as the designated value holder for the `msgText` event parameter. A drawback of such an approach is that, in general, it leads to code fragmentation, where a single conceptual task can be specified in various different (and not predetermined) places, potentially significantly reducing code readability. For that reason, we thought it was better to have all the code and specification in one place, even if the user has to write some JavaScript.

4.5 Dynamic Template-Based Rendering Engine

To go along with this declarative approach for programming the core business logic of an event-based system, RED implements a mechanism for declaratively building graphical user interfaces. The main responsibility of this mechanism is to automatically and efficiently update and re-render the GUI (or relevant parts of it) when a change is detected in the data model. This idea is similar to the concept of "data bindings" (e.g., [52, 56]), but is more general and more flexible.

```
/* ------------ record signatures ------------ */
var AuthUser   = Red.mk_record("AuthUser");
var User       = Red.mk_record("User", AuthUser);
var Msg        = Red.mk_record("Msg");
var ChatRoom   = Red.mk_record("ChatRoom");
var AuthClient = Red.mk_record("AuthClient");
var AuthServer = Red.mk_record("AuthServer");
var Client     = Red.mk_record("Client", AuthClient);
var Server     = Red.mk_record("Server", AuthServer);

/* ------------ event signatures ------------ */
var Register   = Red.mk_event("Register");
var SignIn     = Red.mk_event("SignIn");
var SignOut    = Red.mk_event("SignOut");
var Unregister = Red.mk_event("Unregister");
var CreateRoom = Red.mk_event("CreateRoom");
var JoinRoom   = Red.mk_event("JoinRoom");
var SendMsg    = Red.mk_event("SendMsg");

/* ------------ record meta ------------ */
ChatRoom.meta = new Red.Model.RecordMeta({
 "name"      : "ChatRoom",
 "short_name": "ChatRoom",
 "sigCls"    : ChatRoom,
 "abstract"  : false,
 "parentSig" : Red.Model.Record,
 "subsigs"   : [],
 "fields"    : [
   new Red.Model.Field({
    "parent"      : ChatRoom,
    "name"        : "name",
    "type"        : "String",
    "multiplicity": "one" }),
   new Red.Model.Field({
    "parent"      : ChatRoom,
    "name"        : "members",
    "type"        : User,
    "multiplicity": "set" }),
   new Red.Model.Field({
    "parent"      : ChatRoom,
    "name"        : "messages",
    "type"        : Msg,
    "multiplicity": "seq",
    "owned"       : true })]
});
...

/* ------------ event meta ------------ */
SendMsg.meta = new Red.Model.EventMeta({
 "name"      : "SendMsg",
 "short_name": "SendMsg",
 "sigCls"    : SendMsg,
 "abstract"  : false,
 "parentSig" : Red.Model.Event,
 "subsigs"   : [],
 "params"    : [
   new Red.Model.Field({
    "parent"      : SendMsg,
    "name"        : "room",
    "type"        : ChatRoom,
    "multiplicity": "one" }),
   new Red.Model.Field({
    "parent"      : SendMsg,
    "name"        : "msgText",
    "type"        : "String",
    "multiplicity": "one" })]
});
...
```

Figure 6. Excerpt from the JavaScript translation of the domain model, which the client-side code can program against.

[3] Our JavaScript library actually does not complain if a numeric id is passed where a record object is expected—having all the meta-model information available, it can easily find the event parameter by the name, look up its type, and reflectively construct an instance of that type. Instead of using this shortcut (which works only for record objects) in the main text, we used a more verbose version to illustrate a more general approach and all of its power and flexibility.

[4] Note that this dollar sign has nothing to do with the jQuery dollar sign; it is rather our own syntax for recognizing attribute values that should be computed by evaluating the JavaScript code inside `${}`.

Traditionally, GUIs are built by first constructing a basic visual layout, and then registering callbacks to listen for events and dynamically update bits and pieces of the GUI when those events occur. In contrast, we want the basic visual layout (like the one in Figure 4) to be sufficient for a dynamic and fully responsive GUI. In other words, we want to let the designer implement (design) a single static visualization of the data model, and from that point on rely on the underlying mechanisms to appropriately and efficiently re-render that same visualization every time the underlying data changes.

To implement this approach, we expand on the well-known technique of writing GUI widgets as textual templates with embedded formulas (used to display actual values from the data model) and using a *template engine* [7] to evaluate the formulas and paste the results in the final output. To specify input templates, we use the ERB language (the default template language in Rails) without any modifications. Unlike the existing renderer for ERB, however, our system detects and keeps track of all field accesses that happen during the evaluation of embedded formulas. Consequently, the result of the rendering procedure is not a static text, but a *view tree* where embedded formulas are hierarchically structured and associated with corresponding field accesses (as illustrated in Section 4.1). That view tree is what enables the permanent data bindings—whenever the underlying data changes, the system can search the tree, find the affected nodes, and automatically re-render them.

In the context of web applications, only a textual response can be sent back to the client. Therefore, when an HTTP request is received, the associated template is rendered, and a view tree is produced. The view tree is saved only on the server side. The client receives the same plain-text result that the standard ERB renderer would produce along with some meta-data to denote node delimiters; the browser renders the plain-text response, and our client-side JavaScript library saves the meta-data. When a data change is detected on the server-side, the server finds and re-renders the affected nodes and pushes plain-text *node updates* to corresponding clients; each client then, already having the meta-data, knows where to cut and paste the received update to automatically refresh the GUI.

5. Automated Reasoning and Analysis

Although SUNNY simplifies the development of interactive web applications, and by construction eliminates a whole class of concurrency bugs, it does not eliminate all possible bugs. The user implementation of events can still fail to satisfy the functional requirements of the application. Applying the standard software quality assurance techniques to SUNNY programs is, therefore, still of high importance. We designed SUNNY with this in mind, and in this section we discuss how our programming paradigm is amenable to techniques like automated testing, model checking, and software verification.

5.1 Testing

Testing is the most widely used method for checking program correctness. Testing an event-driven system is both challenging and time consuming, because one needs to generate *realizable* traces (sequences of events). The challenging part in discovering realizable traces is that the preconditions need to hold for each event in the sequence, and the time-consuming part is that the traces can be long, and therefore, there can be too many of them to explore manually. Having both preconditions and postconditions of each event formally specified in our event model allows us to use symbolic-execution techniques [36], and build on recent successes in this domain [69], to discover possible traces automatically.

A symbolic execution engine would start with an empty path condition; at each step, the engine would consider all events from the model and discover the ones that are realizable from the current state (this can be done by using an automated SMT solver [12, 19] to check if there exists a model in which both the current path condition and the event's precondition are satisfied). When an event is found to be realizable, a new state is created and the event's postcondition is appended to the path condition for the new state. Since at each step of this process multiple events may be found to be realizable, the algorithm proceeds by exploring the entire graph, effectively yielding a *state diagram* by the end. Figure 7 depicts the state diagram extracted from the running example (Figure 1). Each node in the diagram describes a symbolic state and each edge describes a transition that can happen when the condition on the edge is satisfied and the event is executed. For example, moving from the initial state to the next state requires that a user initiates a `SignIn` event and provides a correct name and password. This transition results in the execution of the postcondition of the `SignIn` event.

In addition to automated testing of traces, a state diagram can be used to automatically create a test environment – the state necessary before the execution of a test – for all unit tests. Considering Figure 1, if a developer wants to test the `SendMsg` event, there should be a registered user in a room. To create such a state, a sequence of other events have to be executed before `SendMsg`. Inferring from Figure 7, `SignIn` and `CreateRoom` event handlers must be executed. Executing these events requires solving the precondition of each event on the path.

Functional unit testing of events also becomes easier. A black-box unit test for the `SendMsg` event would have to check that the message sent indeed gets added to the list of messages of the given room, that it gets added to the very end of the list, that no other messages get dropped from that list, etc. In SUNNY, this can be done directly, without having

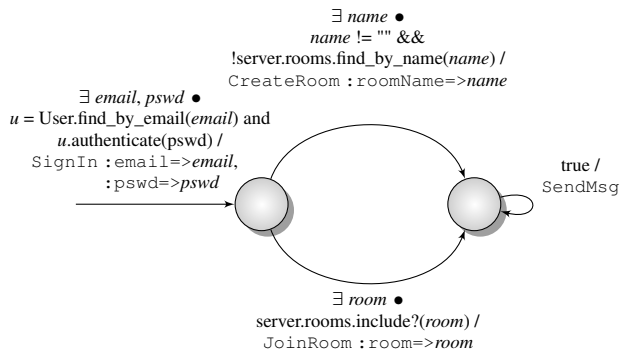

Figure 7. State diagram for the IRC example

to set up any mock objects, e.g., to abstract the network and the actual peer points, as no network is required.

In a traditional event-driven system, an implementation of a single functional unit is often fragmented over several classes. Consider the `SignIn` event: the user sends his or her credentials to the server, the server checks the authenticity, sends back the result, and based on that result, both the server and the client update their local state. In the traditional model, the client-side code can initiate the event (by sending a message to the server), and schedule a continuation that will run when a response is received. The continuation, which is typically a separate procedure, then implements the logic for updating the local state based on the server response. Such fragmented code is very hard to test as a unit, so it is often turned into an integration test, and integration tests are typically more laborious to write and require more elaborate setup. In SUNNY, because of the shared view of the global data, there is no need for such fragmentation; the event handler can be a single procedure that updates only the global data model, meaning that it can easily be tested as a unit.

5.2 Model Checking

Loosely coupled events without explicit synchronization and communication allow model checking to scale. The source of non-determinism in SUNNY models is the order in which events are executed. Because of the non-determinism in scheduling, a model may exhibit different behavior for the same input (i.e., the same values of event parameters) with a different order of event execution. The goal of software model checking is conceptually to explore all orders to ensure the correct execution. Note that the exploration need consider only the semantics of the model and not the semantics of the underlying runtime system. Based on our prior experience with model checking actor programs [39, 65], X10 programs [27], and database applications [26], we believe that an efficient model checking approach can be developed for our new paradigm.

For example, a model checker can be used to check end-to-end properties for all scenarios that the system can possibly exhibit. One such property could be "it is impossible that at one point two different chat rooms have two different users with the same name". The tool can automatically either confirm that the property in question always holds, or find a scenario (i.e., a sequence of events leading to a state) in which the property is violated.

5.3 Verification and Program Synthesis

The technique of discovering realizable sequences of events can also be used to synthesize higher-level operations. For example, a novice IRC user may wonder what are the steps that need to be taken in order to post a message in a chat room. Given such an end-goal, a tool can discover that one possible scenario to achieve that goal is to first `SignIn`, then `JoinRoom`, and finally `SendMsg`. An alternative solution would be to `CreateRoom` instead of `JoinRoom` at the second step. These scenarios can be displayed to the designer and serve as a guide to better understanding possible functionalities of the system (which can be especially useful for bigger systems with many possible events).

6. Discussion

In the previous sections we described several new techniques and concepts this paper proposes to research and develop. In this section we discuss some benefits that directly follow from or are enabled by those techniques.

It can be argued that designing a system around a given property is the best way to ensure that the system correctly implements that property [34]. This paper is certainly in that spirit since it encourages the programmer to carefully design and specify the core part of an event-driven system, i.e., the event model. Furthermore, the programmer does so mostly declaratively, by specifying key properties of events in isolation, without being bogged down by the operational details of the entire distributed system.

We believe that, in most cases, even the event effects (postconditions) might be specified fully declaratively, and yet efficiently executed. We showed previously that declarative specifications can be executable (within a traditional object-oriented language) with certain performance handicaps [48]. Moreover, Near and Jackson [53] showed that, in a setting of a typical web application, most server-side actions (or "events" in our terminology) boil down to (possibly conditional) assignments to variables, which is still declarative, but much easier to execute efficiently. They also showed how this fact can be exploited to build a scalable verifier, which is of comparable complexity to executing a declarative postcondition in the first place.

Our system also lends itself to model-based user interface software tools, which, by definition, take a high-level declarative model of an interactive system and help the programmer build a user interface (UI) for it (either through an integrated development environment or automated code generation) [61]. For example, a UI can be automatically gener-

ated from a SUNNY model that contains generic widgets for querying existing records, creating new instances of records defined in the model, creating associations between existing records via the fields defined in the model, triggering events, and so on, all while respecting the model-defined invariants, event preconditions, and privacy policies. Some existing implementations of scaffolding can already generate a graphical UI that supports standard CRUD operations (create, read, update, delete) for all data model classes; in contrast, with SUNNY models *scaffolding of events* is supported, thus enabling a fully generic user interface that actually covers the *full* functionality of the system.

7. Evaluation

7.1 Comparison with a Web Application in Meteor

We implemented the IRC example in Meteor, a framework designed specifically for fast development of modern, interactive web applications, and compared it to the presented implementation in SUNNY. We make no strong claims based on this simple case study; we only quantify the effort needed to develop this example (in terms of the number of lines of code) and report on our experiences using both systems.

SUNNY and Meteor share the idea that a single data model should be used across the application, even in a distributed setting, and that any updates to it should be automatically propagated to all connected nodes. The main difference is in the representation of the shared data. Meteor relies on MongoDB [1], a NoSQL database which stores data as untyped JSON documents, meaning that the database schema is fully dynamic, and can change anytime. In contrast, models in SUNNY are strongly typed, which is essential to achieving a precise code analysis, but also necessary for implementing various tools, such as the GUI builder.

For comparison, our implementation of the IRC example in Meteor is given in Figure 8. The number of lines of code is about the same, but we believe that SUNNY models tend to be more readable because they make much more explicit both conceptual and structural information about the system. Furthermore, because all the concepts in SUNNY models have a precisely defined semantics, these models can serve as a good documentation on their own.

Another consequence of lack of structure in the Meteor code is the tendency to tightly couple business logic and GUI code. For example, events are often directly tied to JavaScript UI events (e.g., lines 6, 24, 44), and their handlers can fetch values directly from the DOM elements (e.g., lines 7, 8, 25, 26).

We believe that our model-based paradigm has a clear advantage over the dynamic NoSQL model when it comes to applying tools and techniques for various code analyses. In other words, Meteor is mainly focused on providing a platform where data updates are automatically propagated to relevant clients; we are also concerned about the software engineering aspects of the system, its overall design, correctness, testability, and analyzability, as described in Section 5. Most of the ideas from that section would be difficult to apply to Meteor programs.

```
1  Rooms = new Meteor.Collection("rooms");
2
3  if (Meteor.isClient) {
4   // Create Room
5   Template.irc.events({
6    'click input.createRoom': function () {
7     var roomName = $("#roomName").val();
8     var userName = $("#userName").val();
9     // Ignore empty names
10    if (roomName) {
11     var room = Rooms.findOne({name: roomName});
12     if (room == undefined) {
13      Rooms.insert({name: roomName, creator: userName,
14                    members: [userName], messages: []});
15      Session.set("userName", userName);
16      Session.set("user_id", userName);
17     }
18    }
19   }
20  });
21
22  // Join Room
23  Template.irc.events({
24   'click input.joinRoom' : function () {
25    var roomName = $("#roomName").val();
26    var userName = $("#userName").val();
27    // Check if room exist
28    var room = Rooms.findOne({name: roomName});
29    if (room != undefined) {
30     // Check if name is taken
31     var userRoom = Rooms.findOne({
32      members: { $in: [userName] }});
33     if (userRoom == undefined) {
34      Rooms.update({_id: room._id},
35                   {$push: {members: userName}});
36      Session.set("userName", userName);
37     }
38    }
39   }
40  });
41
42  // Send a Message
43  Template.irc.events({
44   'click input.send': function () {
45    var userName = Session.get("userName");
46    // Create a message to be sent
47    var message = Session.get("userName") + ": " +
48          $("#message").val() + "\n";
49    var room = Rooms.findOne({
50     members: { $in: [Session.get("userName")] }});
51    Rooms.update({_id: room._id},
52                 {$push: {messages: message}});
53   }
54  });
```

Figure 8. Implementation of the IRC example in Meteor.

7.2 Comparison with a Client-Server System in Java

In this section, we quantify the effort it took us to build a relatively simple real-time, multi-player game in Java, and discuss how using a technology like SUNNY would significantly simplify certain steps in the process. In fact, the challenges we encountered while developing this game actually inspired the SUNNY project.

SNAP'N'SHOT [2] is a twist on paintball. In paintball, players carry paint guns and shoot one another by firing paint bullets. In SNAP'N'SHOT, players carry cell phones and shoot one another by taking pictures. The game targets the

Android platform and is implemented entirely in Java as a client-server system.

The main challenge developing this game was establishing a solid architecture for concurrent event processing and real-time client notification. The effort to manually implement a message passing backbone, synchronize accesses to shared data, maintain connections alive, and keep all the clients updated resulted in 4000 lines of Java code, as well as several tough concurrency bugs along the way.

All that effort could be reduced to writing a simple model in SUNNY, similar to the one we used for the IRC example. SNAP'N'SHOT defines events that are equivalent to `CreateRoom`, `JoinRoom` and `SendMsg` (except that they are called `CreateGame`, `JoinGame`, and `ShotFired`); its data model also matches that of IRC quite closely.

We have implemented a prototype of SUNNY for client-server Java programs (communicating over sockets), but we have yet to retrofit the implementation of SNAP'N'SHOT to use the new technology.

8. Related Work

8.1 Event-Driven Programming

There are two main styles of building distributed systems: (1) asynchronous or event-driven, and (2) using threads and locks. There has been a lot of debate over whether one is superior to the other. Dabek et al. [17] convincingly argue that event-driven systems lead to more robust software, offer better performance, and can be easily programmed given an appropriate framework or library.

There exist many frameworks or libraries designed to support the event-driven programming paradigm. They are all similar to ours in that they provide an easy, event-driven way of writing distributed applications. Meteor, previously discussed, is one such library; another popular one is *Node.js* [66]. They eliminate the need to manually manage threads and event queues, but typically do not provide an abstract model of the system, amenable to formal analysis.

Approaches like *TinyGALS* [14] and ESP* [64], which focus on programming embedded systems, also provide special support for events. The *TinyGALS* framework additionally offers a structured model of the whole system and also implements global scheduling and event handling. It uses code generation to translate models into an executable form, unlike ESP* which embeds the Statechart [31] concepts in a high-level general-purpose (Java-like) language. ESP* mainly focuses on correctly implementing the Statechart semantics.

Tasks [23] provides language support for complex tasks, which may consist of multiple asynchronous calls, to be written as a single sequential unit (procedure), without having to explicitly register callback functions. This is achieved by a translation of such sequential procedures to a continuation-passing style code. Tasks is not concerned with specifying the top-level event model of the system, and is orthogonal to our framework.

The *implicit invocation* mechanism [25] provides a formal way of specifying and designing event-driven systems. Events and the bindings of events to methods (handlers) are decoupled and specified independently (so that the handler can be invoked "implicitly" by the runtime system). This provides maximum flexibility but can make systems difficult to understand and analyze. In our framework, we decided to take the middle ground by requiring that one event handler (the most essential one, the one that implements the business logic of the system by updating the core data model) is explicitly bound to an event.

Functional Reactive Programming [21] is a programming paradigm for working with mutable values in a functional programming language. Its best known application is in fully functional, declarative programming of graphical user interfaces that automatically react to changing values, both continuous (like time, position, velocity) and discrete (also called events). Implementations of this paradigm include Elm [16] (a standalone language that compiles to HTML, CSS and JavaScript) and Flapjax [45] (an implementation embedded in JavaScript, designed specifically to work with Ajax). Our approach to self-updating GUI can also be seen as a specific application of functional reactive programming.

8.2 Data-Centric Programming

Another, increasingly popular, method for specifying distributed data management is using Datalog-style declarative rules and has been applied in the domain of networking (e.g., Declarative Networking [42], Overlog [41]), distributed computing (e.g., the BOOM project [10], Netlog [29]), and also web applications (e.g., Webdamlog [9], Reactors [22], Hilda [70], Active XML [8]). The declarative nature of Datalog rules makes this method particularly suitable for implementing intrinsically complicated network protocols (or other algorithms that have to maintain complex invariants); manually writing an imperative procedure that correctly implements the specification and respects the invariants is a lot more difficult in this case.

By contrast, we focus on applications that boil down to simple data manipulation in a distributed environment (which constitutes a large portion of today's web applications), and one of our goals is to provide a programming environment that is easy to use by even non-expert programmers who are already familiar with the object-oriented paradigm.

8.3 Code Generation and Program Synthesis

The idea of using increasingly higher-level abstractions for application programming has been a common trend since the 1950s and the first Autocoder [28] systems which offered an automatic translation from a high-level symbolic language into actual (machine-level) object code. The main argument is that software engineering would be easier if pro-

grammers could spend their time editing high-level code and specifications, rather than trying to maintain optimized programs [11]. Our approach is well aligned with this idea, with a strong emphasis on a particular and widely used domain of web application programming.

Executable UML [44] (xUML) also aims to enable programming at a high level of abstraction by providing a formal semantics to various models in the UML family. Model-driven development approaches based on xUML (e.g., [46, 47]) translate the UML diagrams by generating code for the target language, and then ensure that the diagrams and the code are kept in sync. Our system is conceptually similar, and it also follows the model-driven development idea, but instead of using code generation to translate models (diagrams) to code, we want to make models first-class citizens and to have an extensive framework that implements the desired semantics by essentially interpreting the models at runtime (an actual implementation may generate and evaluate some code on the fly to achieve that). Minimizing the amount of auto-generated code makes the development process more convenient, as there is no need to regenerate the code every time the model changes.

Similar to code generation, the main goal of program synthesis is also to translate code from a high-level (often abstract, declarative) form to a low-level executable language. Unlike code generation, however, a simple translation algorithm is often not sufficient; instead, more advanced (but typically less efficient) techniques (e.g., search algorithms, constraint solving, etc.) have to be used. The state of the art in program synthesis focuses on synthesizing programs from various descriptions, e.g,. sketches [63], functional specifications [37], input-output pairs [32], graphical input-output heaps [62], or first-order declarative pre- and post-conditions [40].

The core of our framework is a little further from the traditional program synthesis techniques; although it does aim to provide a high-level surface language for specifying/modeling various aspects of the system (events, privacy policies, GUI templates), it does not perform any complex search-based procedure to synthesize a piece of code. Given the declarative and formal nature of our models, however, program synthesis is still relevant to this work, as it might be applied to implement some advanced extensions, e.g., to synthesize higher-level operations from basic events (as briefly discussed in Section 5).

8.4 Declarative Privacy Policies

In their most general form, policies are used to map each *user* (*subject*), *resource*(*object*) and *action* to a decision, and are consulted every time an action is performed on a resource by a user [38]. In our framework, resources correspond to fields, actions correspond to field accesses[5], and the user is the entity executing the action.

Systems for checking and ensuring privacy policies are typically based either on Access Control Lists (ACL) or Information Flow (IF). ACLs attach a list of *permissions* to concrete *objects*, whereas IF specifies which flows (e.g., data flowing from variable `x` to variable `y`) are allowed in the system. In both cases, when a violation is detected, the operation is forbidden, for example by raising an exception.

Our security model is more in the style of access control lists, in the sense that we attach policies to statically defined fields (as opposed to arbitrary pieces of data), but it has a flavor of information flow as well, since we automatically check all data flowing to all different machines and ensure that no sensitive information is ever sent to a machine that does not have required permissions (which, in our system, means that there is no policy that explicitly restricts that access). Similar to the access modifiers in traditional object-oriented languages (e.g., `private`, `protected`, `public`, etc.), our model also focuses on specifying access permissions for various fields. However, the difference is that our permission policies are a lot more expressive and more flexible than static modifiers, and can also depend on the dynamic state of the program. In addition, they are completely decoupled from the data model, so the policies can be designed and developed independently.

Information flow systems either rely on sophisticated static analysis to statically verify that no violating flows can exist (e.g., Jif [50, 51]), or dynamically labeling sensitive data and tracking where it is flowing (e.g., RESIN [72] or Dytan [15]). Unlike most other information flow systems, Jeeves [71] allows policies that are specified declaratively and separately from the rest of the system, and instead of halting the execution when a violation is detected, it relies on a runtime environment to dynamically compute values of sensitive data before it is disclosed so that all policies are satisfied. This approach is similar to our serialization technique when we automatically hide the sensitive field values before the data is sent to a client.

Margrave [18, 24, 54] implements a system for analyzing policies. Similar to our system, Margrave policies are declarative and independent of the rest of the system (which they call "dynamic environment"). Their main goal, however, is to statically analyze policies against a given relational representation of the environment, and to check if a policy can be violated in any possible (feasible) scenario, whereas we are only interested in checking field accesses at runtime. To enable efficient analysis, the Margrave policy language is based on Datalog and is more restrictive than the first-order logic constraints that we allow in our policies.

[5] Our policy language currently does not allow differentiating between reads and writes, but it could; we will consider adding that extension if we encounter examples where that distinction proves to be necessary.

Attribute-based access control (ABAC) adds attributes ($name \rightarrow value$ pairs) to any entity in the system (e.g., user, resource, subject, object, etc.) so that policies can be expressed in terms of those attributes rather than concrete entities. Our system can be viewed as an instantiation of this model: our fields can be seen as attributes, machines as subjects, and records as resources; both records and machines can have fields, and policies are free to query field values. Many other ABAC systems have been designed and implemented (e.g., [49, 67, 73]), each, however, using somewhat different model from the other. Jin et al. [35] recently proposed a formal ABAC model to serve as a standard, and used it to express the three classical access control models (discretionary [59], mandatory [58], and role-based [60]).

8.5 GUI Builders

Our dynamic template engine for building graphical user interfaces, combines two existing techniques: *data binding* and *templating*.

Data binding allows select GUI widget properties to be bound to concrete object fields from the domain data model, so that whenever the value of the field changes, the widget automatically updates its property. Changes can optionally be propagated in the other direction as well, that is, when the property is changed by the user, the corresponding field value gets updated simultaneously.

Templating, on the other hand, takes a free-form text input containing a number of special syntactic constructs supported by the engine which, at the time of rendering, get dynamically evaluated against the domain data model and get inlined as strings in the final output. Such constructs can include embedded expressions (formulas), control flow directives (if, for loops, etc.), or, in a general case, arbitrary code in the target programming language. This adds extra flexibility, as it allows generic programming features to be used in conjunction with static text, enabling widgets with dynamic layouts to be defined.

Even though existing data binding implementations (e.g., WPF and their textual UI layout language XAML [52] for .NET, UI binder [56] for Android, JFace [30] for Java, Backbone [55] for JavaScript) allow for textual widget templates, those templates are typically allowed to contain only simple embedded expressions (e.g., a path to an object's field), only at certain positions in the template (to provide bindings only for select widget properties). No control structures are allowed, which makes it difficult to design a widget that chooses from two different layouts depending on the state of the application. Conversely, existing template engines (e.g., ASP [43] for .NET, Haml [4] and ERB [3] for Ruby, FreeMarker [68] for Java) provide all that extra flexibility, but do not preserve data bindings, making it difficult to push changes to the client when the model changes.

In this work, we combine these two techniques, to achieve the flexibility of generic template engines and still have the luxury of pushing the changes to the clients and automatically re-rendering the UI. The main reason why that makes the problem more difficult than the sum of its parts is the fact that formulas in the template can evaluate to arbitrary elements of the target language (e.g., HTML), including language keywords, special symbols, tag names, etc. This is unlike the existing UI frameworks with data-bindings, where all bindings are assigned to (syntactically strictly defined) widget properties.

9. Conclusion

Advances in web frameworks have made it much easier to develop attractive, featureful web applications. Most of those efforts are, however, mainly concerned with programming servers and their clients in isolation, providing only a set of basic primitives for intercommunication between the two sides, thus imposing a clear boundary. We believe that there is an entire class of web applications, and distributed programs in general, for which that boundary can be successfully erased and removed from the conceptual programming model the programmer has to bear in mind. SUNNY is a generic programming platform for developing programs that fall into that class.

Acknowledgments

This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-1138967, CCF-1012759, and CCF-0746856. We would like to thank anonymous reviewers for their thoughtful comments on the draft of this paper.

References

[1] MongoDB home page. `http://www.mongodb.org/`.

[2] SNAP'N'SHOT home page. `http://www.snapnshot.me/`.

[3] Ruby's native templating system. `http://ruby-doc.org/stdlib-1.9.3/libdoc/erb/rdoc/ERB.html`.

[4] Haml template engine for Ruby. `http://haml.info`.

[5] Meteor - Pure JavaScript web framework. `http://meteor.com`.

[6] Ruby on Rails web framework. `http://rubyonrails.org/`.

[7] Template engine for web applications. `http://en.wikipedia.org/wiki/Template_engine_%28web%29`.

[8] S. Abiteboul, O. Benjelloun, and T. Milo. Positive active XML. In *Proceedings of the Symposium on Principles of Database Systems*, pages 35–45, 2004.

[9] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for Web data management. In *Proceedings of the Symposium on Principles of Database Systems*, pages 293–304, 2011.

[10] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. Hellerstein, and R. Sears. Boom analytics: exploring data-centric,

declarative programming for the cloud. In *Proceedings of the European Conference on Computer Systems*, pages 223–236, 2010.

[11] R. Balzer, T. E. Cheatham, Jr., and C. Green. Software technology in the 1990's: Using a new paradigm. *IEEE Computer*, 16(11):39–45, 1983.

[12] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the International Conference on Computer Aided Verification*, pages 515–518, 2004.

[13] F. P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley, 1995.

[14] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: a programming model for event-driven embedded systems. In *Proceedings of the Symposium on Applied Computing*, pages 698–704, 2003.

[15] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[16] E. Czaplicki. Elm: Concurrent FRP for functional GUIs. 2012.

[17] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the SIGOPS European Workshop*, pages 186–189, 2002.

[18] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *Proceedings of the International Joint Conference on Automated Reasoning*, pages 632–646, 2006.

[19] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the International Conference on Computer Aided Verification*, pages 81–94, 2006.

[20] J. Edwards. Coherent reaction. In *Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 925–932, 2009.

[21] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the International Conference on Functional Programming*, pages 263–273, 1997.

[22] J. Field, M. Marinescu, and C. Stefansen. Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications. *Theoretical Computer Science*, 410 (2):168–201, 2009.

[23] J. Fischer, R. Majumdar, and T. Millstein. Tasks: Language support for event-driven programming. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation*, pages 134–143, 2007.

[24] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the International Conference on Software Engineering*, pages 196–205, 2005.

[25] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of the Formal Software Development Methods*, pages 31–44, 1991.

[26] M. Gligoric and R. Majumdar. Model checking database applications. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 549–564, 2013.

[27] M. Gligoric, P. C. Mehlitz, and D. Marinov. X10X: Model checking a new programming language with an "old" model checker. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 11–20, 2012.

[28] R. Goldfinger. The IBM type 705 autocoder. In *Papers presented at the Joint ACM-AIEE-IRE Western Computer Conference*, pages 49–51, 1956.

[29] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. *Proceedings of the International Conference on Practical Aspects of Declarative Languages*, pages 88–103, 2010.

[30] J. Guojie. *Professional Java Native Interfaces with SWT/JFace*. Wiley, 2006.

[31] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[32] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 317–328, 2011.

[33] D. Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.

[34] D. Jackson, M. Thomas, L. I. Millett, et al. *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, 2007.

[35] X. Jin, R. Krishnan, and R. Sandhu. A unified attribute-based access control model covering DAC, MAC and RBAC. In *Proceedings of the Data and Applications Security and Privacy*, pages 41–55. 2012.

[36] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[37] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 316–329, 2010.

[38] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1): 18–24, 1974.

[39] S. Lauterburg, M. Dotta, D. Marinov, and G. A. Agha. A framework for state-space exploration of Java-based actor programs. In *Proceedings of the International Conference on Automated Software Engineering*, pages 468–479, 2009.

[40] K. R. M. Leino and A. Milicevic. Program extrapolation with Jennisys. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, pages 411–430, 2012.

[41] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Operating Systems Review*, volume 39, pages 75–90, 2005.

[42] B. Loo, T. Condie, M. Garofalakis, D. Gay, J. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM*, 52(11): 87–95, 2009.

[43] M. MacDonald. *Beginning ASP.NET 4.5 in C#*. Apressus Series. Apress, 2012.

[44] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Object Technology Series. Addison-Wesley, 2002.

[45] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, pages 1–20, 2009.

[46] D. Milicev. *Model-Driven Development with Executable UML*. Wrox Programmer to Programmer. Wiley, 2009.

[47] D. Milićev. Towards understanding of classes versus data types in conceptual modeling and UML. *Computer Science and Information Systems*, 9(2):505–539, 2012.

[48] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *Proceedings of the International Conference on Software Engineering*, pages 511–520, 2011.

[49] T. Moses et al. Extensible access control markup language (XACML) version 2.0. *Oasis Standard*, 2005.

[50] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[51] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[52] A. Nathan. *WPF 4: Unleashed*. Sams, 2010.

[53] J. P. Near and D. Jackson. Rubicon: Bounded verification of web applications. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[54] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave tool for firewall analysis. In *Proceedings of the International Conference on Large Installation System Administration*, pages 1–8, 2010.

[55] A. Osmani. *Developing Backbone.js Applications*. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2013.

[56] J. Ostrander. *Android UI Fundamentals: Develop & Design*. Pearson Education, 2012.

[57] D. Rayside, V. Montaghami, F. Leung, A. Yuen, K. Xu, and D. Jackson. Synthesizing iterators from abstraction functions. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 31–40, 2012.

[58] R. S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[59] R. S. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[60] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[61] E. Schlungbaum. Model-based user interface software tools current state of declarative models. Technical report, Graphics, visualization and usability center, Georgia institute of technology, GVU tech report, 1996.

[62] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proceedings of the Symposium on the Foundations of Software Engineering*, pages 289–299, 2011.

[63] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, 2006.

[64] V. C. Sreedhar and M.-C. Marinescu. From statecharts to ESP: Programming with events, states and predicates for embedded systems. In *Proceedings of the International Conference on Embedded Software*, pages 48–51, 2005.

[65] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Proceedings of the International Conference on Formal Techniques for Distributed Systems*, pages 219–234, 2012.

[66] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to build high-performance network programs. *Internet Computing, IEEE*, 14(6):80 –83, 2010.

[67] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *Proceedings of the Workshop on Formal Methods in Security Engineering*, pages 45–55, 2004.

[68] N. Willy. *Freemarker*. Culp Press, 2012.

[69] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proceedings of the International Conference on Software Engineering*, pages 611–620, 2011.

[70] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *Proceedings of the International Conference on Data Engineering*, pages 32–32, 2006.

[71] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 85–96, 2012.

[72] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the Symposium on Operating Systems Principles*, pages 291–304, 2009.

[73] E. Yuan and J. Tong. Attributed based access control (ABAC) for web services. In *IEEE International Conference on Web Services*, 2005.