Role Definition Language (RDL):
A Language to Describe Context-Aware Roles

Chris Masone
Chris.Masone@dartmouth.edu
Senior Honors Thesis
Advisor: David Kotz
May 31, 2002

Dartmouth College Technical Report TR2002-426

Abstract

As wireless networks become more prevalent, a widening array of computational resources becomes available to the mobile user. Since not all users should have unrestricted access to these resources, a method of access control must be devised. In a context-aware environment, context information can be used to supplement more conventional password-based access control systems. We believe the best way to achieve this is through the use of Context-Aware Role-Based Access Control, a model in which permissions are assigned to entities called *roles*, each principal is a member of one or more roles, and a role's membership is determined using context information. We designed and implemented RDL (Role-Definition Language), a simple, expressive and somewhat extensible programming language to facilitate the description of roles in terms of context information.

1 Introduction

Wireless data connectivity is increasingly available to a widening array of devices and in many more places. As more and more users go wireless, it will make more and more sense to make resources, both computational and otherwise, available to them when they are mobile. Networked devices, such as printers or webcams, could be made available as well as less traditional non-computational resources, such as X-10[1] enabled lights or stereos. Not all users deserve access to every resource at all times, however; in many cases the set of users deserving access depends on the context of the users or of the resource. Static access control would be inappropriate; access should be determined dynamically, in response to changes in the current

[1] http://www.x10.org

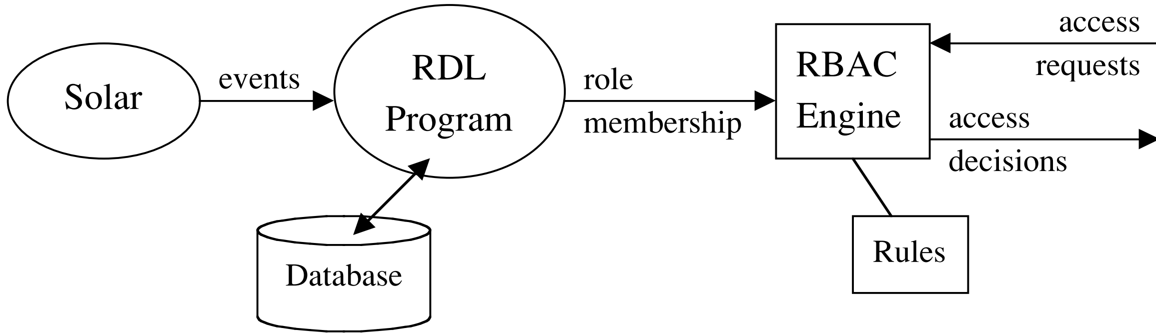

Figure 1: Illustration of a theoretical RBAC system, using Solar and RDL-generated programs

context. *Context* is the situation of the user and devices involved, as well as the surrounding environment. Others have suggested using Context-Aware Role-Based Access Control to implement this behavior. Instead of assigning permissions to a given principal, permissions are assigned to a separate entity called a *role*, and then principals are swapped in and out of this role based on context information [3,4]. This approach requires some way to define the conditions that must be met for a principal to become a member of a given role, as well as some infrastructure to collect the necessary context information. For the latter, we propose to use Guanling Chen's Solar system [1], which delivers context information as a stream of *events* to subscribing entities. For the former, we propose Role-Description Language (RDL), a simple programming language that allows one to write a piece of software that receives context events and, based on programmer-defined conditions, outputs the members of each known role. Any role-based access-control system can then enforce limits on resource access, using rules defined in terms of these roles. Figure 1 demonstrates the place of RDL software in an RBAC system. An RDL program receives events from Solar and uses them to determine role membership.

In the next section of this paper, we provide necessary background information about Solar and RBAC. In Section 3, we provide a description of the syntax and semantics of RDL, followed by an evaluation of the language in Section 4. Section 5 discusses our current implementation of an RDL compiler, explaining the high points of our design and the tools we used. In Section 6 we discuss related projects and their relationship to RDL, and finally, in Section 7, we discuss potential extensions to RDL and improvements to our compiler.

2 Background

RDL cannot be understood without some knowledge of the Solar system and RBAC.

2.1 The Solar System

We use the Solar system [1] to collect and operate on context information. The Solar system is an event-oriented software infrastructure that allows the details of sensor data to be abstracted away from applications using that data. It also supports the transformation and aggregation of data as it travels from the sensors to the application.

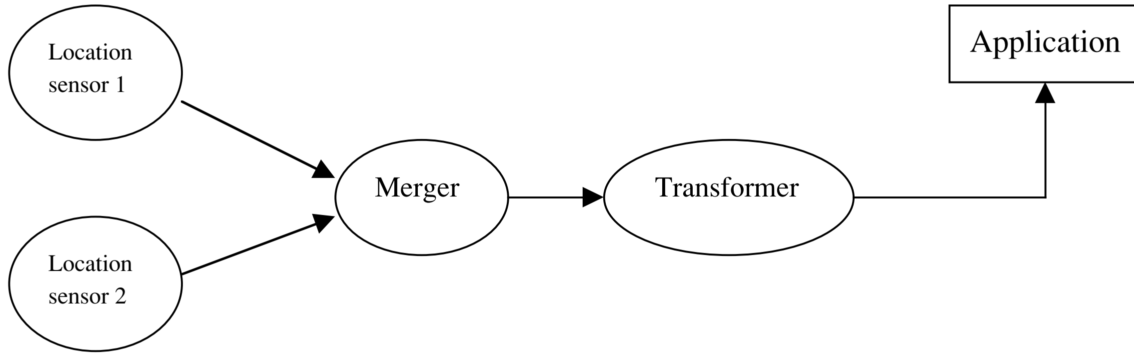

Figure 2: Simple sample Solar operator graph.

Solar uses an operator-graph abstraction, in which sensors are modeled as *sources*, and applications as *sinks*. Each source publishes a stream of *events* containing context information, and the *operator* nodes in the graph between a source and a sink can operate on the incoming streams of events to generate a new stream of events for output. Sources, applications, and operators are all Java objects. The RDL compiler produces several Java files that the programmer compiles and deploys as nodes onto the Solar framework.

Figure 2 shows a simple Solar operator graph. The two location sensors generate raw sensor data and publish event streams to which the Merger subscribes. The Merger publishes a new event stream containing all the events it receives from *both* location sensors. The Transformer subscribes to this new stream, and changes the data of each event in some way, perhaps mapping a sensor number to a room name. The application then subscribes to the resulting event stream. This approach abstracts the details of the sensor data, and allows extensible pre-processing of the events delivered to the application. RDL programs are like Solar applications, receiving events already processed into the necessary format.

2.2 Role-Based Access Control (RBAC)

The basis of RBAC is the concept of a *role* [2]. A *role* is a mechanism for grouping subjects (as principals are referred to in RBAC literature) based on properties of the subject. Since RBAC is commonly used in corporate settings, such properties usually include job title or user responsibilities, such as *Financial Director*, or a more descriptive classification like *people allowed to sign purchase orders*. Permissions in an RBAC system are associated with roles, and then subjects are assigned to roles. Thus, RBAC provides a layer of indirection allowing a security administrator to avoid managing subjects and their permissions individually. In a system where subjects need to change permissions often, this is a much more convenient way to view access control. Roles are similar to groups, a common entity in access control, and groups can be used to implement roles, but there is a conceptual difference between the two. A subject may be a member of many groups, and have all the permissions associated with all those groups. A subject may also be a member of many roles, but not all of the roles may be simultaneously *active*. The subject only has the permissions afforded by his active roles. Some systems constrain the user such that he can only activate one role at a time, thus enforcing a "separation of duties" principle.

3 RDL Semantics

Conceptually, RDL-produced software works like a state machine. Events arrive, event handlers use the events to update the "state of the world", and then roles are defined in terms of this state. We chose this model because it works nicely with our role-definition syntax, which is based on mathematical set notation. When one defines a set in mathematics, one assumes that the "state of the world" is known; in our case, that state can change, and the set may change with it. So, we allow a set to be defined in terms of conditional statements about the state of the world and then track the state and update the set whenever the state changes.

We chose mathematical set notation for our role-definition syntax because it is familiar, which enhances RDL's ease of use. We use an SQL database to store the state of the world because, in many cases, the compiler can pass conditional statements from the programmer's role definitions straight through to the database. Additionally, SQL "join" operations make supporting existentially quantified variables, one of RDL's most important features, nearly trivial. Some other projects [4] have chosen to model their state as a series of Prolog facts and then use a Prolog-like syntax to define their roles. We feel that SQL and mathematical set notation are familiar to more people than Prolog, and therefore make RDL easier to use than a Prolog-based language.

The rest of RDL is also designed to be similar to familiar languages, most notably C and SQL. We also maximize expressivity by pairing RDL with Solar. Thus, the programmer can write some set of Solar operators and build a tree that transforms raw sensor data into a useful format. Then, he can use RDL to write a program that receives events from the Solar tree, stores the information in a database, and define roles through queries on the database. This approach adds the functionality of a query-style interface to context information without the scaling problems inherent in an approach that stores all raw sensor data in a large central database and performing queries on that. By using Solar to distill the data into the most appropriate form possible before storing it into a database, we reduce space requirements and move all the complicated data reduction out into the network.

Compiling an RDL program produces several Solar operators that are then deployed onto the Solar framework along with whatever subscription tree is required to generate the events that the program expects to receive. The most important of these operators is the RoleInfoUpdater, which is the operator that creates the context-information database and uses incoming events to update the database. The compiler also produces an operator corresponding to each role that the programmer declares. Each of these publishes an event containing the current membership of its associated role whenever the database changes. As an example, consider a simple situation in which the programmer has defined two roles, Role A and Role B. The part of the Solar operator graph generated by running the RDL compiler on such an example is shown in Figure 3. Each time the RoleInfoUpdater receives an event from Solar, it updates the database and publishes an event instructing the Role operators to consult the database to determine their new membership.

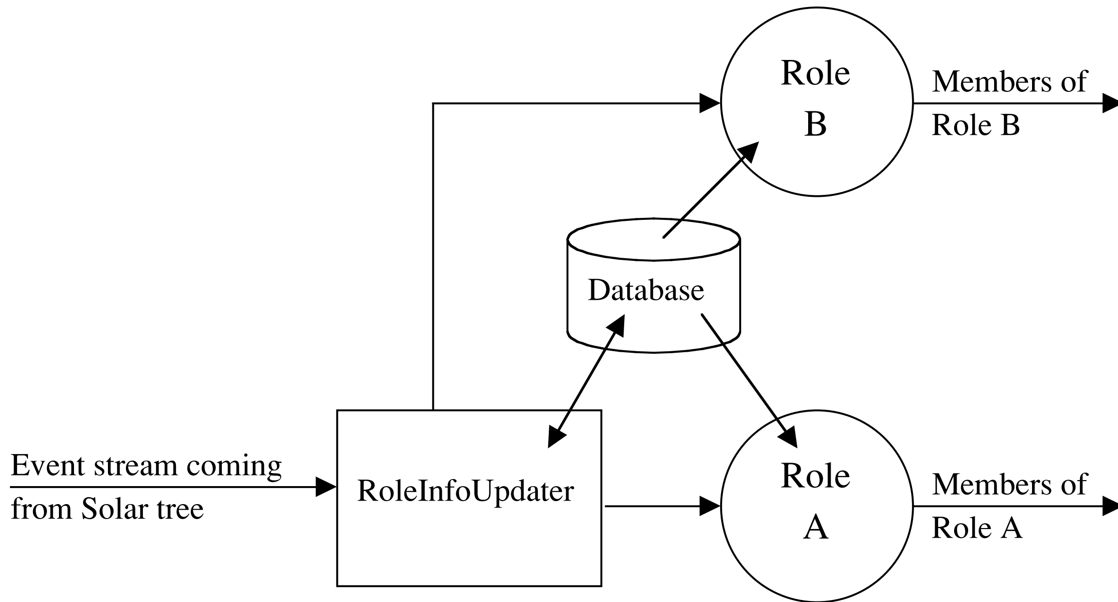

Figure 3: Connections between RDL compiler generated entities.

3.1 RDL Structure

Explication of the structure of the database RDL programs use to store context info requires a basic understanding of the structure of RDL itself. An RDL program is generally made up of 4 files, each of which has its own syntax:

1) the class definition file (.cdf), which allows the programmer to define data structures, referred to as objects, that are analogous to structs in C,
2) the event definition file (.edf), which allows the programmer to define event handlers for each kind of event that can come from the underlying Solar framework,
3) the set definition file (.sdf), which allows the programmer to define sets of objects based upon context information, and
4) the role declaration file (.rdf), which allows the programmer to specify which sets will be exported as roles.

Appendix A lists the full syntax. Below, we introduce RDL to through a series of examples.

3.1.1 Class Definition

Classes in RDL are analogous to structs in C/C++, little more than a set of named and typed fields. RDL natively supports integer, boolean and string datatypes, but the programmer can also define their own typenames using a C-style *typedef* statement. Classes are also valid types for fields in a class declaration, as in Example 1. An instantiation of an RDL class is an

```
typedef string name;
typedef int time;

class Principal {
  index name username;
  int badge_num;
  Room loc;
  int age;
  bool button_pressed;
}

class Room {
  index name roomname;
  list Principal people;
  string size;
  bool light_status;
}
```

Example 1: Sample Class Definition (.cdf) file for RDL

object, which is familiar to any Java or C++ programmer. As shown in the definition of the Room class, RDL also supports lists, though only lists of objects; this constraint arises from the way we implement classes and objects in the database. Each class definition becomes a table in the underlying database, and each object is thus a record in the corresponding table. Each object is also assigned a unique internal identifier, which is then used to refer to that object when lists and set definitions are being evaluated. In the above case, a field named `PrincipalID` wouldbe added to the Principal table, and one called `RoomID` to the Room table.

Since classes are translated into tables, lists become relationships between tables in the database. In the example above, the Principal class has a one-to-many relationship with the Room class (each Room contains many Principals, but each Principal can be in only one Room at a time), which the programmer indicates by including a field with type Room in the definition of Principal and a list of Principals in the Room definition. We implement this relationship by putting the unique identifier of the appropriate record in the Room table into the field named `loc` in every record in the Principal table. So, the field `people` in the Room class, which is a list of Principal objects, is actually represented implicitly by the contents of the field `loc` in the Principal class. Figure 4 demonstrates. Currently, there is no mechanism for associating `people` with `loc`; the relationship is inferred. As a result, the RDL compiler supports only one relationship between any two tables. Rectifying this deficiency should be addressed in future work.

Since it would not make any sense for operators outside of those produced by the RDL compiler to know about objects in the database and their unique identifiers, RDL must provide some mechanism for the programmer to discuss objects in terms of context information coming from Solar. The `index` keyword that appears in the class definitions above, coupled with `infer` statements (discussed below), allow the programmer to get the ID of a particular object and use it to update fields in other objects. If a field is declared as an `index` field, it must be

Principal					
PrincipalID	**username**	**badge_num**	**button_pressed**	**age**	**loc**
0	'cmasone'	85577	0	21	1
1	'dfk'	85566	1	0	1
2	'hawblitz'	85555	0	0	0

Room			
RoomID	**roomname**	**size**	**light_status**
0	'Room 007'	'small'	1
1	'Room 116'	'small'	1
2	'Room 002'	'large'	0

Figure 4: The expression of lists in the database.
This is an example database from a program using the above class definitions. The `people` field of the `Room` class is actually omitted from the database, and whenever a programmer makes a reference to it, the code is converted to refer to the `loc` field of the `Principal` class instead.

unique from object to object so that the programmer can use the value of that field to distinguish them. For instance, in the above class definitions, each Principal contains a field called `loc` of type `Room`. Obviously, events coming in from Solar will not contain the proper RoomID to insert into a Principal's `loc` field; in the above case, the programmer assumes that the events contain room names, so the `roomname` field is an `index` for the Room class. This designation allows him to `infer` the `RoomID` of each Room object given a value for `roomname`. Since the programmer designs the operators and events for the Solar tree that feeds the RoleInfoUpdater, it is reasonable to assume this knowledge.

3.2.2 Event Definition

Before one can write an RDL program, one must first write (or otherwise obtain) the Java operators and events that make up the Solar tree that will feed events to the RoleInfoUpdater. The event-definition portion of RDL allows the programmer to describe the events that the RoleInfoUpdater should expect to see, and to define an event handler for each different class of incoming events. Currently, RDL only supports events described by a set of typed fields, and the only types it supports in events are int, boolean and string. Event handlers are also limited, capable only of updating records or inserting records into the database. This limitation is not that severe, however, because all complex data processing can be done in the Solar tree that feeds the RoleInfoUpdater. Thus, the RoleInfoUpdater receives events containing attribute-value pairs. In Example 2, the programmer describes an event class (an actual Java class) called `PrincipalLocEvent`, which has several attributes with the types listed. He uses an `infer` statement to get the `RoomID` of the Room object whose `roomname` field equals the value of the `roomname` attribute of the event. In an event definition, the syntax used to refer to attributes of the event is a dollar sign followed by the attribute name. This syntax allows attributes of the

```
event PrincipalLocEvent {
  int badge_num;
  bool button_pressed;
  string username;
  string roomname;
  infer Room loc WHERE roomname=$roomname;
} onevent {
  IN Principal {
    WHERE username = $username {
      SET button_pressed=$button_pressed;
      SET loc = $loc;
    } ELSE {
      INSERT username,badge_num,button_pressed,loc
        VALUES $username,$badge_num,$button_pressed,$loc;
    }
  }
}
```

Example 2: Sample Event Definition (.edf) file for RDL

event to be distinguished from fields of an object that have the same name. The need for this syntax is evident in the example shown here, both in the `WHERE` clause of the `infer` statement as well as the event handler code that follows. The `infer` statement shown above results in the creation of an attribute called `loc` that contains the `RoomID` of the record in the Room table whose `roomname` field matched the `roomname` attribute of the event. The `WHERE` clause of an `infer` must be a test for equality between the `index` field of the object being inferred and a field in the event. If no such record exists, one is added and the `RoomID` of that record is placed in `loc`. This new record is initialized using the expression in the `WHERE` clause of the `infer` statement. In Example 2, the `roomname` field of the new `Room` record would be initialized with the value of the `roomname` attribute of the incoming event. "Inferred" attributes can then be used like a normal attribute in the event-handling code in the `onevent` block, below the description of the event's attributes.

The `onevent` block of an event definition defines the event handler for the associated class of event. An `onevent` block contains one or more `IN` blocks, each describing the operations that should take place on a single table. Each `IN` clause is then made up of one or more `WHERE/ELSE` blocks. The `WHERE` clause is almost exactly like an SQL `WHERE` clause; it allows the programmer to specify an expression that must be true for a given record for that record to be updated. If the `WHERE` clause is true for at least one record, the `SET` statements inside the `WHERE` are executed on these records. Otherwise, the `INSERT` statements inside the `ELSE` block are executed on the table specified in the enclosing `IN` clause. The `ELSE` block can be empty if the programmer so desires. It may seem that this syntax is redundant, because in many cases the information in the `INSERT` statement could be inferred from the `SET` statements in the `WHERE` clause, but this is not necessarily the case. In Example 2, the proper treatment of the badge_num field is unclear based on the `SET` statements. Should badge_num be initialized? If so, with what value? Thus, the `WHERE/ELSE` syntax is necessary. At this time, it is not

possible to remove records from any table, though allowing SQL `REMOVE` statements as well as `INSERT` statements in the `ELSE` block would be trivial. Allowing arbitrary SQL in both the `WHERE` and `ELSE` blocks would be more difficult than simply adding `REMOVE` statements, but it would be difficult to allow arbitrary SQL and still provide the typechecking that the RDL compiler currently performs on the contents of the whole `IN` clause.

3.2.3 Set Definition

Set definition is the heart of RDL. A role in RDL is no more than a set of Principal objects, though one can define sets of any kind of objects. The syntax for set definition is roughly similar to the mathematical method of expressing sets. Example 3, for instance, reads "the set 'LightsOn' equals all Rooms *r*, such that *r.light_status* equals true".

```
Room LightsOn() = { Room r |
      ( ( (r.light_status = true) ) )
}
```

Example 3: A simple RDL set definition.

The parentheses after the name of the set are meant to support parameters in set definitions, but this feature has not yet been implemented. The "extra" parentheses in the body of the definition are necessary because RDL currently supports only fully parenthesized boolean expressions in disjunctive normal form, i.e., `(((a=b) && (b=c)) || ((a=c) && (c=d)))`.

RDL has two other features: existentially quantified variables and the usage of previously defined sets in a set definition.

Existentially quantified variables allow the programmer to express criteria in terms of relationships between objects. For example, to express co-location, one could define two Principal objects, *p* and *q* and then define a set as all Principals *p* such that *p.loc* equals *q.loc*.

```
Principal Co-Located() = { Principal p | Principal q
      ( ( (p.loc = q.loc) ) )
}
```

Example 4: An RDL set definition featuring an existentially quantified variable.

As is shown above, existentially quantified variables are declared just after the vertical bar. There can be more than one existentially quantified variable in a single definition, and they can be of different types, though they must all be objects. Additionally, one can compare fields of objects of different types, as in the following definition:

```
Principal ButtonLights() = {
  Principal p | Principal q, Room r
    ( ( (p.loc = q.loc) &&
        (p.button_status = r.light_status) ) )
}
```

Example 5: An RDL set definition featuring a multiple existentially quantified variables.

```
Room LightsOn() = { Room r |
      ( ( (r.light_status = true) ) )
}

Principal Atnd() = { Principal p | Principal q, Room r
  (((p.loc = r) && (r.size = 'big') && (r in LightsOn()))
 ||
   ((p.loc = q.loc) && (q.username = 'dfk')))
}
```

Example 6: An RDL set definition featuring a multiple existentially quantified variables and the use of a set definition inside a later set definition.

If the types match, a field of one object can be compared to an existentially quantified variable. For instance, *p.loc* could be compared directly to a Room *r*. This feature makes more sense when considered in light of RDL's other main syntactic feature: the usage of previously defined sets in later set definitions.

RDL allows the programmer to define sets for the sole purpose of using them in the definitions of other sets, although recursive definitions are not supported. The `in` operator can then be used to test for membership in a set, as in Example 6. The set `Atnd` is the set of all Principals *p*, such that there exists a Principal *q* and a Room *r* for which *p.loc* equals *q.loc* and *q.username* equals 'dfk', or *p.loc* equals *r*, *r.size* equals 'big' and *r* is a member of `LightsOn`. In English, `Atnd` is all Principals that are either co-located with dfk or who are in a big room with the lights on. This (admittedly silly) example uses all the features of RDL that we have discussed so far, and also shows a definition that contains multiple clauses.

3.2.4 Role Declaration

The simplest part of RDL, the role declaration file, simply tells the RDL compiler which sets should be made into roles, that is, which sets need a Solar operator to be generated so that their membership will be published as Solar events. As we said in the previous section, RDL does not yet support parameterized sets, so the syntax of a role declaration is simple:

```
role Attendee = Atnd();
```

Example 7: An RDL role declaration.

Since it would not make sense to allow a set of Room objects, or some other object that does not include information about users, to be used in a role declaration, the RDL compiler currently requires that the programmer define a Principal class with a field called "username" of type string (or some user-defined type that resolves to string). This way, a role operator can just query the database, asking for the username fields of the records that satisfy its query, and publish that list as a Solar event.

4 RDL Evaluation

In this section, we subjectively evaluate the semantics and syntax of RDL before discussing the prototype RDL compiler and the performance of RDL-produced Solar operators in section 5.

4.1 Ease of Use

RDL was designed to be syntactically similar to parts of other languages that are used to express similar ideas. For instance, the set definition portion of RDL was designed to be syntactically similar to mathematical set notation. For programmers comfortable with set notation, writing set definitions in RDL should be pretty easy. Also, since set definitions get compiled into one or more SQL SELECT statements, expressing sets in this way lends itself to the creation of WHERE clauses for those SELECTS; in many cases, the programmer's code can be directly sent to SQL. The syntax of event-definition files has similar advantages; since event handlers are compiled into a series of SQL statements that update the RoleInfoUpdater database, the syntax there was designed to be SQL-like as well. This choice not only makes programming easier, because the language looks familiar to any SQL user, but it also shortens compile time by enabling the compiler to directly reuse code.

The main flaw in terms of use is that one cannot refer to fields of fields; that is, the syntax *p.loc.size*, where *p* is a Principal as defined above and *p.loc* is a Room, is not valid. The reason for this deficiency is that SQL does not support such syntax, so implementing this would have required significantly more work, and the time was not available. We are confident that this flaw could be remedied in the next version of the RDL compiler, probably by creating an internal variable of some kind and expanding the expression. For example, the expression *p.loc.size* = 'big' could be expanded to *p.loc* = *r* && *r.size* = 'big', where *r* is a Room.

4.2 Expressivity

We believe that the fact that the RDL-generated operators sit at the root of a Solar operator tree adds a great deal to the expressive capabilities of our language. Because complicated data transformation can take place out in the operator tree and the results then forwarded to the RoleInfoUpdater, a wide variety of context information can be accommodated by RDL. Provided that one can construct an operator tree to reduce the information to ints, booleans and strings, the resulting events can be used by RDL. Additionally, objects provide a convenient and familiar way to group and organize the context information. The most notable failing of RDL at this time is its inability to express lists of simple data types, but, again, we are confident that this could be resolved in the next version. This feature would be more difficult to implement than chaining of the "." operator; perhaps a two-column table would be created in the database for each list, mapping an internal object IDs to literal values.

The next subsection provides a complete example of an RDL program.

4.3 Example

Class Definition:

```
typedef string name;
typedef int time;

class Room {
  index name roomname;
  list Principal people;
  list Meeting meetings;
}

class Principal {
  index name username;
  Room loc;
  Meeting expected_meeting; # a reference to the next meeting this
                            # principal will attend
}

class Time {
  index int externalID; # allows us to overwrite this one record
  time now;             # instead of creating a new one each time
                        # perhaps a new syntax should be created to
                        # handle classes where there will only be a
                        # single instantiation
}

class Meeting {
  index string mtg_name;
  Room mtg_room;
  list Principal expected_people;
  name chairname;
  time start_time;
  time end_time;
}
```

Event Definition:

```
event TimeEvent {
  int new_time;
} onevent {
  IN Time {
    WHERE externalID = 1 {
      SET now = $new_time;
    } ELSE {
      INSERT now, externalID VALUES $new_time, 1;
    }
  }
}
```

```
event PrincipalLocEvent {
  string username;
  string roomname;
  infer Room loc WHERE roomname=$roomname;
} onevent {
  IN Principal {
    WHERE username=$username {
      SET loc = $loc;
    } ELSE {
      INSERT username, loc VALUES $username, $loc;
    }
  }
}

event MeetingCreationEvent {
  string mtg_name;
  int start;
  int end;
  string chair;
  string roomname;
  infer Room mtg_room WHERE roomname = $roomname;
} onevent {
  IN Meeting {
    WHERE mtg_name = $mtg_name {
      SET mtg_name=$mtg_name,start_time=$start,end_time=$end,
            chairname=$chair,mtg_room=$mtg_room;
    } ELSE {
      INSERT mtg_name,start_time,end_time,chairname,mtg_room
        VALUES $mtg_name,$start,$end,$chair,$mtg_room;
    }
  }
}

event PersonalPlannerEvent {
  string username;
  string next_meeting;
  infer Meeting m WHERE mtg_name = $next_meeting;
} onevent {
  IN Principal {
    WHERE username = $username {
      SET expected_meeting = $m;
    } ELSE {
      INSERT username, expected_meeting VALUES $username, $m;
    }
  }
}
```

Set Definition:

```
# the chair of a meeting is any Principal p such that there exists a
# Time t and a Meeting m such that p is in m's meeting room, p is
# listed as the chair of m, and t tells us that it is currently between
# the start and end times of m
Principal Chair() = {
      Principal p | Time t, Meeting m
      (
       ((p.loc = m.mtg_room) && (p.username = m.chairname) &&
        (t.now >= m.start_time) && (t.now <= m.end_time))
      )
}

# an attendee of a meeting is any Principal p such that there exists
# Principals q and s, Meeting m and Time t such that p is in m's
# meeting room, q is a chairperson and is in m's meeting room, and t
# tells us that it is currently between the start and end of m
# OR
# p is expected at the meeting m, q is a chair and is in m's meeting
# room, and t tells us that it is currently between the start and end
# of m
Principal Attendee() = {
      Principal p | Meeting m, Principal q, Principal s, Time t
      (
       (
        (p.loc = m.mtg_room) &&
        (q in Chair()) &&
        (q.loc = m.mtg_room) &&
        (t.now >= m.start_time) &&
        (t.now <= m.end_time)
       )
      ||
       (
        (p.username = s.username) && # p.username in m.expected_people
        (s in m.expected_people) &&  # is not supported, so this syntax
                                      # must be used instead.
        (q in Chair()) &&
        (q.loc = m.mtg_room) &&
        (t.now >= m.start_time) &&
        (t.now <= m.end_time)
       )
      )
}
```

Role Declaration:

```
role people_at_a_meeting = Attendee();
role chair_of_a_meeting = Chair();
```

5 Implementation

The prototype RDL compiler is written using JavaCC[2], the Java Compiler Compiler, a tool available from WebGain that helps one write compilers in Java. JavaCC allows the compiler developer to convert a formal description of a programming language (such as Appendix A) into a set of Java functions, one for each non-terminal in the language. These functions can contain arbitrary Java code and can also take parameters. We used this functionality to allow us to parse the input files directly into our internal data structures, without having to build an intermediate representation of the input, such as a parse tree.

To manage the context information database, we use a MySQL server running on a well-known remote host. To facilitate our communication with the server, we use the MM MySQL package of Java Database Connectivity (JDBC) drivers[3]. Currently, there is only one database serving all running RoleInfoUpdaters. This configuration is less than ideal, as it could cause conflicts between classes with the same name but different definitions, in addition to providing a potential performance bottleneck. Unfortunately, time constraints made this compromise the only viable solution at this time. Finding some way to differentiate tables from different RoleInfoUpdaters, or even to share common tables, is a subject for future work.

The prototype RDL compiler only generates the RoleInfoUpdater, but this ability is enough to allow us to test the correctness of RDL-generated event handlers and role-membership queries. All sets are modeled as tables in the database, named after the set. Each record in set tables is just an ID of a record in another table. Thus, when another set definition needs to check a value for membership in a particular set, the RoleInfoUpdater simply selects all records in the appropriate database and checks against the returned values.

Appendix B shows a portion of the code produced by the RDL compiler for the example shown in Section 4.3.

6 Related Work

There is surprisingly little work on access-control in context-aware systems. Kazuhiro Minami [5] discusses a method for using context-aware roles in controlling access to context information gathered by the Solar system. RDL is a necessary extension for defining roles and tracking role membership.

Georgiadis et al. [3] combine context information with their team-based access control to create a system called C-TMAC. Their work focuses mostly on determining the actual permissions of a user, and then using context information to determine whether or not the user is allowed to exercise those privileges. Upon login to the C-TMAC system, the user chooses a role to occupy from the set of roles he is allowed to assume. C-TMAC also defines entities called *teams*, which have associated with them certain permissions and certain contexts within which

---

[2] http://www.webgain.com/products/java_cc/

[3] http://mmmysql.sourceforge.net/

those permissions may be exercised. A context is defined by several ranges of values, one for each possible parameter used to describe context; for example, if context is described in terms of a patientID, a location and a time, ranges of acceptable values must be defined for each parameter. So, the user's total set of permissions and valid contexts is calculated from their chosen role and chosen teams, and then context information determines whether the user is allowed to exercise his permissions on a request by request basis. C-TMAC is focused on implementing permissions, a problem that RDL ignores. Thus, the expressivity of C-TMAC's role-definition language is limited. For instance, C-TMAC seems to have no way to express a requirement for co-location of two users, a construct that is trivial in RDL.

The environmental role work, by Covington et al. [4], also has a wider focus than our work; it covers everything from role definition to permission assignment to policy definition. Their project models environmental conditions as roles, which has the advantage of keeping things simple by basing everything on the role abstraction, but it seems to restrict what they can express. There does not seem to be a way to discuss the attributes of a particular principal (or subject, in their language) in relation to those of other principals. It is unclear how concepts like co-location are expressed, and the language does not seem to be as extensible as RDL. It is simple, using a Prolog-style logical syntax for both role and policy definition, but the trade-off of simplicity for expressivity and extensibility may not be the best.

7 Future Work

The RDL compiler exists and, as shown above, produces operators that work efficiently and correctly. Several features remain to be implemented that would enhance both the expressiveness and the simplicity of RDL. Chief among these is parameterization of sets. If one wanted to define two similar sets, such as "people in room 005" and "people in room 003", in the current implementation one would need to write two definitions that are essentially the same, except for one string literal. If the definitions could take parameters, however, one could write a single definition for both, and then just pass in the appropriate string to differentiate the two. This addition is probably the most complex, requiring some coherent naming scheme for all the tables representing the same set definition, but with different parameters. It is related to the issue of differentiating, and possibly reusing, tables from different RoleInfoUpdaters; both require some way of naming tables so that two tables based on the same definition are given different, but easily identifiable, names. Second, to enhance the expressivity of RDL, lists of native data types should be implemented and the limitation on lists of objects should be removed. As we mention above, this modification would also be fairly difficult, probably requiring the addition of some more tables to the database. Finally, as a performance enhancement, only the role operators affected by the last database update should re-evaluate and re-publish the membership of their role. Currently, all role operators do this every time the database is updated, which is a waste of bandwidth and places an unnecessary burden on the database server. This modification would be fairly easy, simply requiring that some information be kept about which fields in which tables

were updated, and then checking that against each definition to see which definition depends on the updated fields.

8 Summary

In response to the need to control access to resources in a wirelessly networked, context-aware environment we have implemented RDL, a language used to define context-aware roles as part of a role-based access-control system. We designed RDL to work with a Solar infrastructure that handles both collection of context-aware data and complicated data-reduction tasks, so that RDL can be as simple as possible. RDL's syntactic similarity to other, well-known programming languages helps make it easier to use, and the programs written with it perform efficiently. We believe that RDL provides a simple, extensible and expressive way to define and discuss context-aware roles as part of a role-based access control system.

9 Acknowledgements

First of all, I would like to thank my advisor, Professor David Kotz, for his guidance and sufferance of my vacillating interests. He tolerated my at times wayward sense of where I wanted to go with this project, and allowed me to keep my own pace, within the bounds of reason. I also thank Kazuhiro Minami and Guanling Chen for their continual feedback on the project, as well as their constant willingness to help me find tools and other resources to help me on my way. I also thank my girlfriend for her understanding, even though I had to go work on my thesis the night of our one-year anniversary.

References

[1] Guanling Chen and David Kotz. Supporting adaptive ubiquitous spplications with the Solar system. Computer Science Technical Report TR2001-397. Dartmouth College, May 31, 2001.

[2] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein and Charles E. Youman. Role based access control models. In *IEEE Computer*, volume 2, February 1996.

[3] Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind K. Dey, Mustaque Ahamad and Gregory D. Abowd. Securing context-aware applications using environment roles. In *Proceedings of the Sixth ACM Symposium on Access control models and technologies,* Chantilly, VA, 2001.

[4] Christos K. Georgiadis, Ioannis Mavridis, George Pangalos and Roshan K. Thomas, Flexible team-based access control using contexts. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies,* Chantilly, VA, 2001.

[5] Kazuhiro Minami and David Kotz. Controlling access to pervasive information in the "Solar" system. Computer Science Technical Report TR2002-422. Dartmouth College, February 28, 2002.

Appendix A: Grammar for RDL

Class Definition:

```
<start> := <typedefs> <classdefs>
<classdefs> := (<classdef>)+
<classdef> := "class" <identifier> "{" (<fielddec>)+ "}"
<fielddec> := <type> <identifier> ";"
<type> := ["list" | "index"] (<builtintype> | <userdeftype>)
<builtintype> := "int" | "bool" | "string"
<userdeftype> := <identifier>
<typedefs> := (<typedef>)*
<typedef> := "typedef" <builtintype> <identifier> ";"
```

Event Definition:

```
<start> := <eventdef>+
<eventdef> := "event" <identifier> "{" <fields> "}" "onevent" "{"
                  (<inclause>)+ "}"
<fields> := <fielddec> (<infer_or_field>)*
<infer_or_field> := <fielddec> | <infer>
<infer> := "infer" <objtype> <fieldname> "WHERE" <inferwhereclause>
<fielddec> := <type> <fieldname> ";"
<fieldname> := <identifier>
<inclause> := "IN" <identifier> "{" (<whereelse>)+ "}"
<whereelse> := "WHERE" <whereclause> "{" <updatestmts> "}" "ELSE "{"
                  <initstmts> "}"
<updatestmts> := (<update>)+
<update> := "SET" <setclause> ";"
<initstmts> := (<init>)+
<init> := "INSERT" <initfields> "VALUES" <initvalues> ";"
<initfields> := <identifier> ("," <identifier>)
<initvalues> := <sqlexp> ("," <sqlexp>)*
<type> := "int" | "bool" | "string"
<objtype> := <identifier>
<setclause> := <identifier> "=" <sqlexp>
                  ( "," <identifier> "=" <sqlexp> )*
<whereclause> := (<identifier> | <sqlexp>) <relop> <sqlexp>
                  ( "," (<identifier> | <sqlexp>) <relop> <sqlexp> )*
<inferwhereclause> := <identifier> "=" <sqlterm>
<relop> := "=" | "!=" | "<=" | ">=" | "<" | ">"
<sqlexp> := (<sqlterm>)+
<sqlterm> := regexp( ( ["a"-"z","A"-"Z","_","0"-"9","(",")","-
","+","!","$","%","&","*"] )+ )
```

Set Definition:

```
<start> := <defns>
<defns> := (<defn>)+
<defn> := <obj_type> <identifier> "(" <listofvars> ")" "=" "{"
              <objdec> "|" <listofobjs> "(" <orclause> ")" "}"
<object_type> := <identifier>
<listofvars> := [<vardec>] ( "," <vardec> )*
<listofobjs> := [<objdec>] ( "," <objdec> )*
<vardec> := <type> <identifier>
<type> := <builtintype> | <userdeftype>
<builtintype> := "int" | "bool" | "string"
<userdeftype> := <identifier>
<objdec> := <identifier> <identifier>
<orclause> := "(" <andclause> ")" [ "||" <orclause> ]
<andclause> := "(" <clause> ")" [ "&&" <andclause> ]
<clause> := <Lvar> <op> ( <Rvar> | <literal> )
<Lvar> := <identifier> ["." <identifier>]
<Rvar> := <identifier> ["(" <params> ")" | "." <identifier>]
<params> := [<identifier>] ( "," <identifier> )*
<op> := "!=" | "==" | ">=" | "<=" | "<" | ">" | "in"
<literal> := <num> | <string> | <boolconst>
<boolconst> := "true" | "false"
```

Role Declaration:

```
<roledecs> := (<roledec>)+
<roledec> := "role" <identifier> "="
                  <identifier> "(" [<params>] ")" ";"
<params> := (<string> | <num>) ( "," (<string> | <num>) )*
```

Reused:

```
<identifier> := regexp( ["a"-"z","A"-"Z","_"]
                         ( ["a"-"z","A"-"Z","_","0"-"9"] )* )
<num> := regexp( ( ["0"-"9"] )+ )
<string> := regexp( "\""
                    (
                      (~["\"","\\","\n","\r"])
                     |
                      ("\\"
                       (
                          [n","t","v","b","r","f","a","\\","?","'","\""]
                         |
                          "0" (["0"-"7"])*
                         |
                          ["1"-"9"] (["0"-"9"])*
                         |
                          ("0x" | "0X") (["0"-"9","a"-"f","A"-"F"])+
                        )
                       )
                     )*
                      "\""
                    )
```

Appendix B: Sample Code Produced by RDL Compiler

```
public void handleEvent(IEvent ev) {

  Hashtable h = new Hashtable();
  Statement s;
  ResultSet rs;
  int num_results;

  try {
    if((ev.getClass().getName()).equals("PrincipalLocEvent")) {
      PrincipalLocEvent e = (PrincipalLocEvent)ev;
      Matcher escaper;
      String escaped_string;
      escaper = (Pattern.compile("(['\\\\])")).matcher(e.username);
      escaped_string = escaper.replaceAll("\\\\$1");
      h.put("username", "'" + escaped_string + "'");

      escaper = (Pattern.compile("(['\\\\])")).matcher(e.roomname);
      escaped_string = escaper.replaceAll("\\\\$1");
      h.put("roomname", "'" + escaped_string + "'");


      s = con.createStatement();
      rs = s.executeQuery("SELECT RoomID from Room WHERE roomname=" +
            (String)h.get("roomname"));
      num_results = 0;
      while(rs.next()) {
        num_results++;
        if(num_results > 1)
            break;
          h.put("loc", (new Integer(rs.getInt(1))).toString());
        }

        if(num_results==0) { //no results
        s.executeUpdate("INSERT INTO Room (roomname) VALUES (" +
                       (String)h.get("roomname") + ")");
        rs = s.executeQuery("SELECT RoomID from Room WHERE roomname=" +
                           (String)h.get("roomname"));
        num_results = 0;
        while(rs.next()) {
          num_results++;
          if(num_results > 1)
            break;
          h.put("loc", (new Integer(rs.getInt(1))).toString());
        }

      }
```

```
      try {
        int retval = 0;
        retval = s.executeUpdate("UPDATE Principal SET loc=" +
                                 (String)h.get("loc") +
                                 " WHERE username=" +
                                 (String)h.get("username"));
        if(retval == 0) {
          s.executeUpdate("INSERT INTO Principal (username,loc) VALUES ("
                          + (String)h.get("username") + "," +
                          (String)h.get("loc") + ")");
        }
      } catch(SQLException PrincipalLocEventse2) {
      }

    }
  } catch(SQLException se) {
    se.printStackTrace();
  }
}
```