Fine-Grained Access Control with Object-Sensitive Roles*

Jeffrey Fischer, Daniel Marino, Rupak Majumdar, and Todd Millstein

Computer Science Department
University of California, Los Angeles
{fischer,dlmarino,rupak,todd}@cs.ucla.edu

Abstract. Role-based access control (RBAC) is a common paradigm to ensure that users have sufficient rights to perform various system operations. In many cases though, traditional RBAC does not easily express application-level security requirements. For instance, in a medical records system it is difficult to express that doctors should only update the records of *their own* patients. Further, traditional RBAC frameworks like Java's Enterprise Edition rely solely on dynamic checks, which makes application code fragile and difficult to ensure correct.
We introduce Object-sensitive RBAC (ORBAC), a generalized RBAC model for object-oriented languages. ORBAC resolves the expressiveness limitations of RBAC by allowing roles to be parameterized by properties of the business objects being manipulated. We formalize and prove sound a dependent type system that statically validates a program's conformance to an ORBAC policy. We have implemented our type system for Java and have used it to validate fine-grained access control in the OpenMRS medical records system.

1 Introduction

Controlled access to data and operations is a key ingredient of system security. *Role-based access control* (RBAC) [9] is an elegant and frequently-used access control mechanism in which a layer of *roles* interposes between users and access privileges. Roles represent responsibilities within a given organization. Authorizations for resource access are granted to roles rather than to individual users and users are given roles according to their functions in the organization. Users acquire all privileges associated with their roles. The intuition behind RBAC is that roles change infrequently within organizations relative to users, and so associating roles with access privileges ensures a stable and reliable access control policy.

As a concrete scenario, consider a hospital in which users can be doctors or patients. Doctors should be able to view and update their patients' records, and patients should be able to view (but not update) their own records. The RBAC way to represent this policy is to introduce two roles *Doctor* and *Patient*, where the *Doctor* role is allowed to both look up and modify patient records and the *Patient* role is allowed only to look up a medical record. Users are then classified as having the *Doctor* or *Patient* roles and

* This material is based upon work supported in part by the National Science Foundation under grants CCF-0545850 and CCF-0546170.

inherit the corresponding access privileges. RBAC is available in standard enterprise software development environments such as Java's Enterprise Edition (Java EE) [16], which insert runtime role checks whenever a privileged operation is invoked.

This simple example highlights two key limitations of the RBAC model and its usage today:

Lack of expressiveness. The role-based implementation described above does not capture all the constraints of our desired policy. The role-based implementation allows doctors to access and modify *any* patient's record, rather than only their own patients. Similarly, the role-based implementation allows patients to access any other patient's record. One way to solve the problem is to give each user his or her own role, but that would remove the advantages of using roles altogether! Simply put, the RBAC model is not fine-grained enough to express common access control requirements.

As a result of this limitation, programmers may be forced to insert manual access checks that augment the ones provided by systems like Java EE. This manual process is error prone, and it is difficult to ensure that the inserted checks properly enforce the desired policy. Alternatively, a system may only enforce a coarse-grained access control policy but additionally maintain a log of accesses to allow system administrators to detect finer-grained violations *a posteriori*.[1]

Lack of static checking. The reliance solely on dynamic checks in today's RBAC-based systems leads to several problems. First, it is difficult for programmers to ensure that their code properly respects the access control policy. Programmers must manually keep track of what roles must be held when each function is invoked, which depends on the set of privileged operations that can potentially be reached during the function's execution. If a function is ever executed in the wrong environment, the only feedback will be a runtime role failure when a privileged operation is invoked, making the problem difficult to diagnose and fix.

Further, because of the cost of runtime role checks, the checks are often hoisted from the privileged operations themselves to the "entry points" of an application. For example, after user authentication, a single role check could be used to determine which web page to display (e.g., one for doctors and another for patients). However, in this case the programmer must manually ensure the sufficiency of this check for all potentially reachable privileged operations downstream, or else the intended access policy can be subverted.

In this paper we address both of these limitations of the traditional RBAC model and associated frameworks. First, we extend the RBAC model to support fine-grained policies like that of our medical records example above. The basic idea is to allow roles and privileged operations to be *parameterized* by a set of *index* values, which intuitively are used to distinguish users of the same role from one another. A privileged operation can only be invoked if both the appropriate role is held and the role's index values matches the operation's index value.

[1] This was the case in two recent security breaches in the news: unauthorized access to Britney Spears' medical records by employees at UCLA medical center and to Barack Obama's cell phone records by employees at Verizon Wireless.

Our parameterized form of RBAC, which we call *Object-sensitive* RBAC (ORBAC), has a natural interpretation and design in the context of an object-oriented language (Sect. 2). Traditional RBAC policies control access at the level of a class. For example, with Java EE a method `getHistory` in a `Patient` class can be declared to require the caller to hold the *Patient* role. In other words, a user with the *Patient* role can invoke the `getHistory` method on *any* instance of `Patient`. In contrast, ORBAC supports access control at the level of an individual object. For example, `getHistory` can now be declared to require the caller to hold the *Patient*<*this.patientId*> role, where the `patientId` field of `Patient` stores a patient's unique identifier.

Second, we provide a type system that *statically* ensures that a program meets a specified ORBAC policy, providing early feedback on potential access control violations. We formalize our static checker for a core Java-like language (Sect. 3). Since types and roles are parameterized by program values (e.g., `this.patientId`), our static checker is a form of *dependent type system*.

We have implemented our static type system for ORBAC as a pluggable type system for Java in the JavaCOP framework [2]. As with frameworks like Java EE, we leverage Java's annotation syntax to specify the role requirements on method calls, but the JavaCOP rules statically ensure the correctness and sufficiency of these annotations. We have augmented the OpenMRS medical records application [21] with a fine-grained access control policy using ORBAC and have used our JavaCOP checker to statically ensure the absence of authorization errors (Sect. 4).

2 Object-sensitive RBAC

We now overview Object-sensitive RBAC and its associated static type system through a simple medical records example in Java, comparing an implementation using standard RBAC in Java EE with one using ORBAC.

2.1 Role-Based Access Control

An RBAC policy can be described as a tuple (U, R, P, PA, UA) consisting of a set of users U, a set of roles R, and a set of permissions P, together with relations $PA \subseteq P \times R$ giving permissions to roles and $UA \subseteq U \times R$ giving (sets of) roles to users [9]. An access of permission p by user u is *safe* if there exists a role $r \in R$ such that $(u, r) \in UA$ (user u has role r) and $(p, r) \in PA$ (role r has permission p).

Figure 1 shows how this model applies to a `Patient` class for which we wish to protect access. Our simplified class provides a factory method `getPatient`, which retrieves the specified patient from the database, and two instance methods: `getHistory` to return a history of the patient's visits and `addPrescription` to associate a new prescription with the patient.

We can group the users of our application into two groups: doctors and patients. In a typical medical records application, doctors can access the data of their patients and patients can access their own data (e.g., through a web self-service feature). In a standard RBAC model, we can represent these two groups with *Doctor* and *Patient* roles. Java

```
public class Patient {
    private int patientId;

    /* factory method to retrieve a patient */
    @RolesAllowed({"Doctor", "Patient"})
    public static Patient getPatient(int pid) { ... }

    @RolesAllowed({"Doctor", "Patient"})
    public List<String> getHistory() { ... }

    @RolesAllowed({"Doctor"})
    public void addPrescription(String prescription) { ... }
    ...
}

public class PatientServlet {
    void displayHistory(int pid, Request req, Response resp) {
        if (req.isUserInRole("Patient")) {
            if (req.userId != pid) {
                throw new AccessError("Cannot access this patient");
            }
        }
        Patient p = Patient.getPatient(pid);
        List<String> hist = p.getHistory();
        ... code to write html representation of hist to resp ...
    }
}
```

Fig. 1. Standard RBAC version of doctor-patient example

EE supports the specification of an RBAC policy through the `@RolesAllowed` annotation [16]. This annotation is placed on a method definition to indicate the set of roles that have permission to invoke the method. In Fig. 1 we have annotated the `getPatient` and `getHistory` methods to permit users with either the *Doctor* or *Patient* role to call these methods. On the other hand, the `addPrescription` method has been annotated to ensure that only doctors can add a prescription to a medical record.

The Java EE tools, and other application frameworks, enforce an RBAC policy dynamically by inserting runtime checks to verify that the user indeed has at least one of the specified roles when an annotated method is invoked. These checks are supported by standard infrastructure that performs user authentication and queries a database or configuration files to determine role membership.

For example, one might maintain a database of users and the roles granted to each user in an external LDAP server, where it can be managed by an administrator. The first time a user attempts to access a protected application resource (e.g., a web page), he is redirected to a login page. The user is *authenticated* by comparing his credentials

against those stored in the LDAP server. The user's identity and roles are then stored in memory (e.g., in a session context) for use by dynamic access control checks.

Limitations of the RBAC model Consider the `PatientServlet` class of Fig. 1, which accesses a patient's medical record. The `displayHistory` method writes an HTML representation of the patient history to a response stream. To do this, it obtains a `Patient` object using `Patient.getPatient` and then calls its `getHistory` method. Due to the annotations on these methods, the Java EE framework will insert dynamic checks on these calls to ensure that the user has either the *Doctor* or *Patient* role.

Unfortunately, these checks are not sufficient to enforce the desired access control policy. For example, the checks allow any patient to access any other patient's medical record! Therefore, programmers must manually insert additional checks, as shown at the beginning of the `displayHistory` method. A similar check may also be necessary to ensure that a doctor only accesses the records of her own patients. These kinds of checks are very fragile and error-prone — one can easily forget or improperly implement the check on some code path that leads to an invocation of a protected method, resulting in a serious security vulnerability.

Another limitation of traditional RBAC frameworks like Java EE is the reliance solely on dynamic checks, which makes it difficult to statically ensure that application code in fact respects the access policy of a protected class. For example, the programmer must ensure that the `displayHistory` method is never invoked by a user who does not have either the *Doctor* or *Patient* roles. This requirement is completely implicit and can only be understood by examining the implementation of `displayHistory` (and in general the implementations of methods transitively called by `displayHistory`). If a program disobeys the requirement, the programmer will receive no warning about the error, which will instead result in a dynamic access check failure. Such dynamic errors can be difficult to diagnose and fix. Further, if the error is not expected by the calling code, it may result in very unfriendly behavior from the user's perspective (e.g., a Java uncaught exception).

2.2 Object-sensitive RBAC

ORBAC is a natural generalization of the formal model for RBAC defined above. With ORBAC, we define $UA \subseteq U \times R \times I$ to be a ternary relation, in which $UA(u,r,i)$ gives a user u an *indexed role* $(r,i) \in R \times I$, where I is a set of *index values*. Permissions are also indexed, and an access by user u to the *indexed permission* $(p,i) \in P \times I$ is *safe* if there exists a role $r \in R$ such that $(u,r,i) \in UA$ (user u has indexed role (r,i)) and $(p,r) \in PA$ (role r has permission p).

In Fig. 2, we reimplement our example using an ORBAC policy. We use two roles: *Patient* and *DoctorOf*, both of which are parameterized by a patient identifier (a Java integer). A patient is given the *Patient* role for his own identifier, allowing him to access his own record but not those of other patients. A doctor is given a *DoctorOf* role for each of her patients, allowing access to those patients but no others.

Conceptually, classes are now parameterized by a set of role indices, which are part of the class's static type, analogous with ordinary type parameters in Java. These role

```
public class Patient {
    @RoleParam public final int patientId;

    /* factory method to retrieve a patient */
    @Requires(roles={"DoctorOf", "Patient"}, params={"pid", "pid"})
    @Returns(roleparams="patientId", vals="pid")
    public static Patient getPatient(@RoleParam final int pid) { ... }


    @Requires(roles={"DoctorOf", "Patient"},
              params={"this.patientId", "this.patientId"})
    public List<String> getHistory() { ... }

    @Requires(roles="DoctorOf", params="this.patientId")
    public void addPrescription(String prescription) { ... }
    ...
}

public class PatientServlet {
    @Requires(roles={"DoctorOf", "Patient"},
              params={"pid", "pid"})
    void displayHistory(@RoleParam final int pid,
                        Request req, Response resp) {
        Patient p = Patient.getPatient(pid);
        List<String> hist = p.getHistory();
        ... code to write html representation of hist to resp ...
    }
}
```

Fig. 2. ORBAC version of doctor-patient example

indices may then be used in role annotations within the class. While our formalism explicitly parameterizes classes in this way, as shown later, our implementation employs additional annotations to achieve the same effect without modifying Java's syntax. Class role parameters are modeled as public final fields of the class that are declared with the `@RoleParam` annotation. For example, the `@RoleParam` annotation on the `patientId` field of `Patient` indicates that this field will be used as an index in role annotations within the class. The `@RoleParam` annotation can also be used on final formal parameters to achieve the effect of method parameterization, as seen on the `pid` parameter of the `getPatient` method.

Our `@Requires` annotation is analogous to Java EE's `@RolesAllowed` annotation, indicating the set of roles that have permission to invoke the annotated method. To stay within Java's metadata syntax we use two parallel arrays, `roles` and `params`, to specify the roles. For example, the `@Requires` annotation on `getPatient` in Fig. 2 allows only users with either the `DoctorOf<pid>` or `Patient<pid>` role to invoke the method, where `pid` is the patient identifier passed to the method. The `@Requires` annotations on the other methods are similar but they use the `patientId` field of the

receiver as the role index to appropriately restrict access to that `Patient` object. Unlike the `@RolesAllowed` annotation, `@Requires` does not introduce a dynamic check. Instead, all calling code is statically checked to ensure at least one of the required roles is held.

The `@Requires` annotation is a form of method precondition for access control, while our `@Returns` annotation is a form of postcondition. For example, the `@Returns` annotation on `getPatient` asserts that the returned `Patient` object has a `patientId` role parameter field which is equal in value to the patient identifier passed to the method. Our static type system checks the body of the method to ensure the equality between the role parameters holds. The type system can then assume that this equality holds after a call to `getPatient`. In this way, we support modular typechecking for access control.

Resolving the limitations of the RBAC model The `PatientServlet` class of Fig.2 illustrates how ORBAC resolves the limitations identified earlier of the RBAC model. Unlike the version in Fig. 1, no manual access checks are required. These checks are now part of the access control policy and are reflected in the `@Requires` annotations on the methods of `Patient`. Therefore, it is easy for both humans and tools to reason about a program's access control policy just based on program annotations, without examining the bodies of methods.

Further, access control is now statically checked, providing early feedback on possible violations. The `displayHistory` method is annotated with `@Requires`, restricting the method to users of the `DoctorOf<pid>` and `Patient<pid>` roles. With this annotation, the method's body can be statically guaranteed to obey the access control policy of `Patient`. The call to `getPatient` satisfies that method's `@Requires` clause, so the call typechecks. The `getPatient` method's `@Returns` clause indicates that the returned patient object's `patientId` parameter is equal to `pid`, which then allows the call to `getHistory` to typecheck successfully.

Subtle errors are now caught statically rather than dynamically. For example, if the call to `getPatient` in `displayHistory` passed a patient identifier other than `pid`, the call would correctly fail to typecheck, since a patient could be accessing the record of a patient other than himself. Also, the annotation on `displayHistory` in turn allows *its* callers to be modularly checked at compile time, ensuring that they have the necessary roles for the eventual access to `Patient`.

Incorporating dynamic checks Our static type system makes explicit (via the `@Requires` annotation) the precondition that must be satisfied on entry to a method *m* to ensure that the access control policies of all methods transitively called by *m* will be obeyed. We insist that top-level methods (e.g., `main` for a standalone application or `service` for a servlet-based web application) have no `@Requires` annotation. That is, the application's external interface must have no precondition and thus can assume nothing about the roles that the current user holds. In order to allow an unprotected method to call a method protected by a `@Requires` annotation, our type system provides a flexible mechanism for interfacing with the program's authorization and authentication logic through the definition of *role predicate methods*. These methods are identified by the `@RolePredicate` annotation, which also indicates the role that the method tests for. Our static type system incorporates a simple form of flow sensitivity to ensure that

```
public class Request {
    @RolePredicate(roles="Patient", params="pid")
    public boolean hasPatientRole(@RoleParam final int pid) { ... }

    @RolePredicate(roles="DoctorOf", params="pid")
    public boolean hasDoctorOfRole(@RoleParam final int pid) { ... }
}

public class PatientServlet {
    void displayHistory(@RoleParam final int pid,
                        Request req, Response resp) {
        if (!(req.hasPatientRole(pid) ||
              req.hasDoctorOfRole(pid)))
        {
            throw AccessError("Cannot access this patient");
        }
        Patient p = Patient.getPatient(pid);
        List<String> hist = p.getHistory();
        ... code to write html representation of hist to response ...
    }
```

Fig. 3. Use of role predicate methods in `displayHistory`

method calls whose role requirements are not met by the current method's `@Requires` annotation occur only after appropriate dynamic checks succeed.

As a simple example, Fig. 3 contains a new version of `PatientServlet`'s `displayHistory` method that performs the necessary role checks dynamically. The method no longer has a `@Requires` clause, but our static type system recognizes that the method is safe: the dynamic role checks ensure that the calls on the `Patient` class are only reached when the user has the appropriate *Patient* or *DoctorOf* role. Unlike the manual dynamic checks in the standard RBAC example shown earlier, these checks are statically ensured to be sufficient. Any errors in the dynamic checks in Fig. 3 (e.g., accidentally using a patient identifier other than `pid`) will be caught at compile time. Further, the dynamic checks can be placed as early as possible in the execution of an application without the risk that a check will be forgotten on some code path to a protected method.

The role predicate methods are treated as black boxes by our type system. They are free to consult a framework's security infrastructure or to implement authentication and authorization however the application designer sees fit. In fact, a particular predicate method could always return true and be used to achieve an effect similar to J2EE's `@RunAs` annotation, which allows components to be invoked with a security identity other than that of the currently authenticated user. In short, predicate methods provide a flexible mechanism for incorporating the runtime checks that are necessary to ascertain security credentials, and our type system ensures that their use is sufficient to satisfy declared method preconditions.

ClassDecl	K	::=	class $C\langle\overline{r}\rangle\{\overline{T\,f};\overline{M}\}$
MethodDecl	M	::=	$\langle\overline{r}\rangle T\ m(\overline{T}\ \overline{x})$ requires $\Phi\{e\}$
Exprs	e	::=	$x \mid e.f \mid e.m\langle\overline{\rho}\rangle(\overline{e}) \mid$ new $T(\overline{e}) \mid e \Box e \mid$ use Φ in e $\mid$ pack $\rho, e \mid$ unpack e as r, x in e
Vals	v	::=	new $C\langle\overline{i}\rangle(\overline{v}) \mid$ pack i, v
Types	T	::=	$C\langle\overline{\rho}\rangle \mid \exists r.T$
RoleContext	Φ	::=	propositional formula over atoms in Q
Roles	Q	::=	$R\langle\rho\rangle$
Indices	ρ	::=	$r \mid i$
IndexVarContext	Δ	::=	$\cdot \mid \Delta, r$
VariableContext	Γ	::=	$\cdot \mid \Gamma, x : T$

Fig. 4. Grammar for the ORBAC language and type system. Metavariable C ranges over class names, m over method names, f over field names, R over role names, r and q over index variables, i and j over index constants, and x over program variables.

3 Formal Semantics

We have formalized the static and dynamic semantics of a small Java-like language in which ORBAC policies can be expressed and statically checked, and we have proven a type soundness theorem. Figure 4 shows the syntax of our language, a variant of Featherweight Java [14]. Our language models only the core features necessary to study the ORBAC model and its static type system formally. For this reason we have omitted inheritance, although our implementation handles it in the standard way, as described in Sect. 4.1.

In our Java implementation of ORBAC described in the previous section, index variables are specially designated fields and method parameters. In our formal language, we explicitly parameterize classes, methods, and roles using the syntax of Java generics. For greater expressiveness, we include a form of existential types to classify expressions whose role indices are not statically known. This models, for example, the situation in our Java implementation where a method's return type is parameterized by an index, but no information about this index's value is provided (e.g., via a `@Returns` annotation). Expressions of existential type are introduced in our core language by a *pack* expression and eliminated by an *unpack* expression, in the usual way [24]. Our core language includes a `use` expression for dynamically changing the set of held roles, which is a simplified form of the role predicate methods in our Java implementation.[2] Finally, we include a non-deterministic choice construct ($e_1 \Box e_2$) as a simple form of conditional.

Access protection is expressed in our Java implementation using a `@Requires` annotation indicating the set of roles that may invoke a method. This set can be viewed as a *disjunctive* predicate to be satisfied on entry to the method. We provide a more general mechanism in our formal language; methods include a `requires` clause which can specify an arbitrary propositional formula over roles as a precondition for invocation.

[2] The `use` expression can be viewed as a role predicate method that always succeeds. The possibility of a predicate method returning `false` can be modeled by combining `use` with non-deterministic choice. For example, the expression $(\text{use } \Phi \text{ in } e_1) \Box e_2$ models the situation where e_1 is executed if a dynamic check for predicate Φ succeeds, and otherwise e_2 is executed.

$\boxed{K \text{ ok}}$

$$\frac{\overline{r} \vdash \overline{T} \qquad \overline{M} \text{ ok in } C\langle\overline{r}\rangle}{\text{class } C\langle\overline{r}\rangle\{\overline{T\,f};\overline{M}\} \text{ ok}} \tag{C-OK}$$

$\boxed{M \text{ ok in } T}$

$$\frac{\overline{r},\overline{q} \vdash T \qquad \overline{r},\overline{q} \vdash \overline{T} \qquad \overline{r},\overline{q} \vdash \Phi \qquad \Phi;\overline{r},\overline{q};\overline{x}:\overline{T},\text{this}:C\langle\overline{r}\rangle \vdash e:T}{\langle\overline{q}\rangle T\ m(\overline{T}\ \overline{x}) \text{ requires } \Phi\{e\} \text{ ok in } C\langle\overline{r}\rangle} \tag{M-OK}$$

$\boxed{\Phi;\Delta;\Gamma \vdash e:T}$

$$\Phi;\Delta;\Gamma \vdash x:\Gamma(x) \tag{T-VAR}$$

$$\frac{\Phi;\Delta;\Gamma \vdash e:T \qquad \text{fields}(T) = \overline{T}\ \overline{f}}{\Phi;\Delta;\Gamma \vdash e.f_i:T_i} \tag{T-FIELD}$$

$$\frac{\text{fields}(T) = \overline{T}\ \overline{f} \qquad \Phi;\Delta;\Gamma \vdash \overline{e}:\overline{T} \qquad \Delta \vdash T}{\Phi;\Delta;\Gamma \vdash \text{new } T(\overline{e}):T} \tag{T-NEW}$$

$$\frac{\Phi;\Delta;\Gamma \vdash e_1:T \qquad \Phi;\Delta;\Gamma \vdash e_2:T}{\Phi;\Delta;\Gamma \vdash e_1 \sqcap e_2:T} \tag{T-CHOOSE}$$

$$\frac{\Delta \vdash \rho \qquad \Phi;\Delta;\Gamma \vdash e:[r \mapsto \rho]T}{\Phi;\Delta;\Gamma \vdash \text{pack } \rho,e:\exists r.T} \tag{T-PACK}$$

$$\frac{\begin{array}{c} r \notin \Delta \qquad \Gamma(x) \textit{ undefined} \\ \Phi;\Delta;\Gamma \vdash e_1:\exists q.S \qquad \Delta \vdash T \qquad \Phi;\Delta,r;\Gamma,x:[q \mapsto r]S \vdash e_2:T \end{array}}{\Phi;\Delta;\Gamma \vdash \text{unpack } e_1 \text{ as } r,x \text{ in } e_2:T} \tag{T-UNPACK}$$

$$\frac{\Delta \vdash \Phi' \qquad \Phi';\Delta;\Gamma \vdash e:T}{\Phi;\Delta;\Gamma \vdash \text{use } \Phi' \text{ in } e:T} \tag{T-USE}$$

$$\frac{\begin{array}{c} \Phi;\Delta;\Gamma \vdash e:S \qquad \Delta \vdash \overline{\rho} \qquad \text{msig}(S,m) = \langle\overline{r}\rangle\overline{T} \xrightarrow{\Phi'} T \\ \Phi;\Delta;\Gamma \vdash \overline{e}:[\overline{r} \mapsto \overline{\rho}]\overline{T} \qquad \Phi \Rightarrow [\overline{r} \mapsto \overline{\rho}]\Phi' \end{array}}{\Phi;\Delta;\Gamma \vdash e.m\langle\overline{\rho}\rangle(\overline{e}):[\overline{r} \mapsto \overline{\rho}]T} \tag{T-INVK}$$

$\boxed{\text{fields}(T) = \overline{T}\ \overline{f}}$

$$\frac{\text{class } C\langle\overline{r}\rangle\{\overline{T\,f};\overline{M}\} \in \textit{ClassDecls}}{\text{fields}(C\langle\overline{\rho}\rangle) = \overline{[\overline{r} \mapsto \overline{\rho}]T}\ \overline{f}} \tag{FIELDS}$$

$\boxed{\text{msig}(T,m) = \langle\overline{r}\rangle\overline{T} \xrightarrow{\Phi} T}$

$$\frac{\text{class } C\langle\overline{r}\rangle\{\overline{T\,f};\overline{M}\} \in \textit{ClassDecls} \qquad \langle\overline{q}\rangle S\ m(\overline{S}\ \overline{x}) \text{ requires } \Phi\{e\} \in \overline{M}}{\text{msig}(C\langle\overline{\rho}\rangle,m) = \langle\overline{q}\rangle[\overline{r} \mapsto \overline{\rho}]\overline{S} \xrightarrow{[\overline{r} \mapsto \overline{\rho}]\Phi} [\overline{r} \mapsto \overline{\rho}]S} \tag{M-SIG}$$

Fig. 5. Typing rules for our formal language.

The typing rules for our formal language are shown in Fig. 5. Expressions are typechecked under three contexts: Φ is the role context represented as a propositional formula over roles, Δ keeps track of the index variables that are in scope, and Γ is the usual free-variable typing context. The rules depend on a set of simple well-formedness judgments, which ensure that all referenced index variables are in scope. For example, $\Delta \vdash T$ in the premise of T-NEW ensures that the type being constructed does not refer to any undefined index variables.

The most interesting rule is T-INVK which includes a logical entailment check in the premise that guarantees that the current role context Φ satisfies the callee's `requires` precondition after appropriate substitution of actual indices for index parameters. Methods are typechecked modularly by rule M-OK which uses the Φ specified in a method's `requires` clause as the role context when checking the body.

Rules T-PACK and T-UNPACK are standard for existential type systems. The role variable r in rule T-UNPACK is required to be fresh, which matches the intuition that existential types classify objects with unknown index values. An unpacked role variable r can only be employed to satisfy role checks within a `use` statement that grants roles involving r. This is analogous to performing a dynamic role predicate check on an object with an unknown index in our Java implementation. Rules FIELDS and M-SIG only apply to class types, so an existential package must be unpacked before its fields and methods are accessed and values of existential type cannot be directly instantiated.

The dynamic semantics for our formal language is shown in Fig. 6. These evaluation rules perform role checks that model the dynamic checks on privileged operations used in most existing RBAC systems. Our type soundness result, however, establishes that such dynamic role checking is unnecessary for well-typed programs. Like the typing judgment, the evaluation judgment includes a role context. This context is used in rule E-INVK, which performs a dynamic entailment check that the current role context is sufficient to satisfy the method's declared precondition. Rule E-CONGRUENCE steps subexpressions according to the evaluation order established by the evaluation contexts, leaving the role context unchanged. Rule E-USE1 ignores the current role context and dynamically evaluates its subexpression under the specified context.

We have proven a type soundness theorem, which ensures that well-typed programs cannot fail dynamic role entailment checks. The theorem is proven using the standard progress and preservation style [30]. Full details are given in the accompanying technical report [10]; we provide statements of the key results here:

Lemma 1 (Progress) *If* $\Phi;\cdot;\cdot \vdash e : T$*, then either* e *is a value or there is an expression* e' *such that* $\Phi' \vdash e \longrightarrow e'$ *for any* Φ' *where* $\Phi' \Rightarrow \Phi$.

Lemma 2 (Preservation) *If* $\Phi;\Delta;\Gamma \vdash e : T$ *and* $\Phi \vdash e \longrightarrow e'$*, then* $\Phi;\Delta;\Gamma \vdash e' : T$.

These lemmas imply a type soundness theorem as well as the key corollary about role checking:

Theorem 1 (Type Soundness) *If* $\Phi;\cdot;\cdot \vdash e : T$*, then* e *will not get stuck when evaluated under any role context* Φ' *such that* $\Phi' \Rightarrow \Phi$.

Corollary 1 (Dynamic Entailment Checks Unnecessary) *Well-typed programs cannot fail dynamic role entailment checks.*

Evaluation Contexts $E ::= [\,] \mid E.f \mid E.m\langle\overline{\rho}\rangle(\overline{e}) \mid v.m\langle\overline{\rho}\rangle(v,\ldots,E,e,\ldots,e)$
$\mid \text{new } T(v,\ldots,E,e,\ldots,e) \mid \text{pack } \rho, E \mid \text{unpack } E \text{ as } r,x \text{ in } e$

$\boxed{\Phi \vdash e \longrightarrow e}$

$$\frac{\Phi \vdash e \longrightarrow e'}{\Phi \vdash E[e] \longrightarrow E[e']} \quad \text{(E-CONGRUENCE)}$$

$$\frac{\text{fields}(T) = \overline{T}\ \overline{f}}{\Phi \vdash \text{new } T(\overline{v}).f_i \longrightarrow v_i} \quad \text{(E-FIELD)}$$

$$\frac{\text{mbody}(T, m\langle\overline{\rho}\rangle) = (\overline{x}, e) \qquad \text{msig}(T,m) = \langle\overline{\sigma}\rangle\overline{S} \xrightarrow{\Phi'} S \qquad \Phi \Rightarrow [\overline{\sigma} \mapsto \overline{\rho}]\Phi'}{\Phi \vdash \text{new } T(\overline{v}).m\langle\overline{\rho}\rangle(\overline{v'}) \longrightarrow [\overline{x} \mapsto \overline{v'}][\text{this} \mapsto \text{new } T(\overline{v})]e} \quad \text{(E-INVK)}$$

$$\Phi \vdash \text{unpack } (\text{pack } i, v) \text{ as } r,x \text{ in } e \longrightarrow [x \mapsto v][r \mapsto i]e \quad \text{(E-UNPACK)}$$

$$\Phi \vdash e_1 \,\square\, e_2 \longrightarrow e_1 \quad \text{(E-CHOOSE1)}$$

$$\Phi \vdash e_1 \,\square\, e_2 \longrightarrow e_2 \quad \text{(E-CHOOSE2)}$$

$$\frac{\Phi' \vdash e \longrightarrow e'}{\Phi \vdash \text{use } \Phi' \text{ in } e \longrightarrow \text{use } \Phi' \text{ in } e'} \quad \text{(E-USE1)}$$

$$\Phi \vdash \text{use } \Phi' \text{ in } v \longrightarrow v \quad \text{(E-USE2)}$$

$\boxed{\text{mbody}(T, m\langle\overline{\rho}\rangle) = (\overline{x}, e)}$

$$\frac{\text{class } C\langle\overline{r}\rangle\{\overline{T}\ \overline{f}; \overline{M}\} \in \mathit{ClassDecls} \qquad \langle\overline{q}\rangle S\ m(\overline{S}\ \overline{x}) \text{ requires } \Phi\{e\} \in \overline{M}}{\text{mbody}(C\langle\overline{\rho}\rangle, m\langle\overline{\sigma}\rangle) = (\overline{x}, [\overline{q} \mapsto \overline{\sigma}][\overline{r} \mapsto \overline{\rho}]e)} \quad \text{(M-BODY)}$$

Fig. 6. Evaluation for our formal language.

4 Experience: The OpenMRS Case Study

We implemented our ORBAC checker as an extension to Java in the JavaCOP pluggable types framework [2]. To evaluate our approach, we took OpenMRS [21], an existing open source medical records application, and retrofitted it to use an ORBAC policy to protect access to patient data. OpenMRS is implemented in Java using the Spring application framework [28], which is a commonly used alternative to Java EE. Spring provides several useful modules, including an *inversion of control* container, an aspect-oriented programming framework, and integration with the Hibernate framework for persistence [13]. Spring's access control framework supports standard RBAC policies, which can be configured by an administrator.

4.1 Implementation of ORBAC using JavaCOP

Our checker implementation makes use of the annotations `@Requires`, `@Returns`, and `@RolePredicate` that were introduced in Sect. 2.2.

Several practical issues that are not modeled in the formalism are addressed in our implementation. Class inheritance is supported. The checker enforces the standard requirements on method overriding: an overriding method must have a compatible, possibly weaker precondition (`@Requires` annotation) and a compatible, possibly stronger postcondition (`@Returns` clause). Methods without a `@Requires` annotation are considered to have the precondition `true`, so they can be invoked in any context. Hence, methods that override such methods are required to not have a `@Requires` annotation.

While our formalism uses arbitrary propositional formulas for requires clauses, our Java implementation restricts `@Requires` and `@RolePredicate` annotations to be disjunctions of roles. This means that role contexts are formulas in conjunctive normal form (CNF); the `@Requires` clause of a method provides the first conjunct and dynamic role predicate checks add conjuncts to the context. This simplifies typechecking by allowing us to perform a series of subset checks rather than checking arbitrary logical implication.

We make use of JavaCOP's support for flow-sensitive reasoning [17] to implement the static updating of the role context based on role predicate method invocations. JavaCOP's flow framework properly handles Java's complex control flow, including exceptional control flow. As a result, our checker can statically validate the style of dynamic checks used in Fig. 3, as well as many other styles.

The implementation of the checker was fairly straightforward. It contains 174 lines of code in the declarative JavaCOP language and about 450 lines of Java code defining the flow analysis and some supporting functions and data structures.

4.2 OpenMRS architecture

The OpenMRS source contains over 160,000 lines of code, spread over 633 files, not including the frameworks and other infrastructure that it depends upon. Figure 7 shows a simplified UML diagram of some key patient-related classes defined by OpenMRS. Patients are represented by the `Patient` class. Each patient has a number of associated

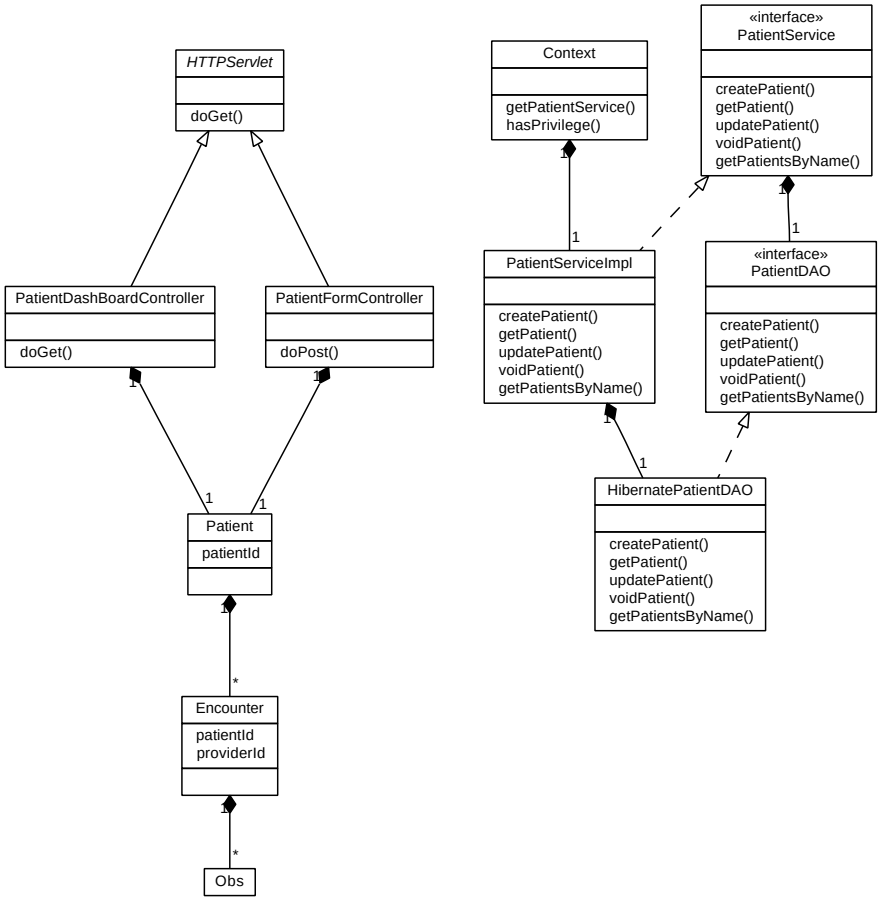

Fig. 7. Patient-related classes in OpenMRS

encounters, each representing a visit to the hospital or clinic. Each encounter may contain multiple *observations* (represented by the `Obs` class) which are used for recording test results and patient vitals.

The OpenMRS application interacts with the client via Java servlets. In Fig. 7, we show the two primary servlets for patients, `PatientDashBoardController`, which renders to HTML a summary of a patient's data, and `PatientFormController`, which accepts a new or updated patient and saves it to the database. These servlets obtain patient records from the database via classes implementing the `PatientService` interface, which defines methods for creating, querying, updating, and voiding patients (as well as many others not shown here). The implementation of `PatientService` is provided by `PatientServiceImpl`, which in turn uses a class implementing `PatientDAO` (DAO stands for "Data Access Object"). The implementation of `PatientDAO` is pro-

vided by `HibernatePatientDAO`, which interacts with the Hibernate framework and isolates Hibernate-specific code.

The patient service implementation `PatientServiceImpl` is made available to servlets via the `Context` class. This class provides static methods for accessing global system state (e.g., mappings between "service" interfaces and their configured implementations) as well as state specific to a given user (e.g., a user id and permissions).

OpenMRS access control framework The implementation of RBAC in OpenMRS adds a level of indirection to the standard RBAC model: methods are protected by assigning required *privileges* through annotations in the code, *roles* are defined as mappings from role names to sets of privileges, and *users* are assigned sets of roles. The role-privilege mapping and the user-role mapping are maintained in the database, permitting them to be changed by an administrator at runtime.

Access policies are configurable in OpenMRS, but the limitations of the RBAC model make it impossible to configure a policy that permits access to a specific object while preventing access to other objects of the same class. In other words, only coarse-grained policies, which restrict access at the level of classes rather than objects, are supported. For example, in one reasonable policy within these restrictions, patients would have no access to the system at all and every healthcare provider would have read-write access to all patients.

Access control requirements are defined using method annotations representing the set of privileges needed to access the method. These annotations are converted to dynamic checks by Spring's aspect-oriented programming framework. For patient data, these annotations are made on the `PatientService` class. There are separate privileges defined for viewing, creating, updating, and deleting patients. The administrator must then assign these privileges to RBAC roles.

Each servlet in OpenMRS may (indirectly) invoke many dynamic privilege checks inserted by Spring. Unfortunately, there is no easy way to tell which privileges are required by a servlet. Changes to the implementation of a servlet may inadvertently change the set of privileges checked in a given situation, leading to runtime errors, which are displayed as an HTML rendering of a Java stack trace.

Privileges may be explicitly checked in the code by calling the `hasPrivilege` method on the `Context` class. These explicit checks are used in situations where authorization occurs in a conditionally executed block or where an implementation needs additional authorization requirements beyond those specified for an interface.

4.3 An ORBAC policy for OpenMRS

With ORBAC we were able to create a new fine-grained access control policy for patient objects, with three roles:

1. Users with the *Supervisor* role have read and write access to all patients. This role is unparameterized — it behaves as a standard RBAC role.
2. Users with the *ProviderFor* role (e.g., doctors) have read and write access to their patients, but not to other patients. This role is parameterized by the patient's id.

User	Assigned roles	Patients allowed read-only access	Patients allowed read-write access	Patients denied access
Alice	Supervisor		Britney, Carol, Dave	
Bob	ProviderFor<Carol>		Carol	Britney, Dave
Britney	Patient<Britney>	Britney		Carol, Dave
Carol	Patient<Carol>, ProviderFor<Britney>	Carol	Britney	Dave
Dave	Patient<Dave>	Dave		Britney, Carol

Fig. 8. Example of access rights for OpenMRS extended with ORBAC

3. Users with the *Patient* role have read access to their own patient record, but not to those of other patients. This role is parameterized by the patient's id.

We only changed the access policies for objects related to patients; other objects in the system are protected by OpenMRS's original RBAC policy.

Example 1. Figure 8 shows an example set of user-to-role assignments and the resulting access rights of these users. There are three patients in the system: Britney, Carol, and Dave. All three have a *Patient* role parameterized by their own id and can thus see, but not modify, their own patient records. Alice holds the unparameterized *Supervisor* role and has read-write access to the three patients. Bob is a provider for Carol, and thus has read-write access to her record, but no access to the other patients. Carol is both a provider for Britney and a patient herself. She does not have read-write access to her own record. □

The mechanism for assigning the *ProviderFor* role turned out to be an interesting design consideration. The OpenMRS database schema and object model implement a one-to-many doctor-patient relationship, so one might consider using the presence of this relationship to grant *ProviderFor* status. However, in a real healthcare environment, multiple doctors and nurses might need to interact with a patient and thus see the patient's record. We chose to base the granting of the *ProviderFor* role on whether there is an encounter record associated with the patient and the provider. This can be determined by an SQL query against the `Encounter` table, the results of which can then be cached to speed up future checks.

The presumed workflow for granting access rights to a patient's data are as follows:

1. When a patient enters the clinic, a user with *Supervisor* access looks up the patient's record, or creates it if necessary.
2. The *Supervisor* selects a doctor to see the patient and then creates an encounter record referencing the patient and the doctor.
3. The doctor now has the *ProviderFor* role for this patient and can update the patient record.

Thus, all the providers who have participated in a patient's care can access the patient record. Other approaches to granting access rights to patient data are possible and enforceable with our pluggable type system.

Implementing the ORBAC policy To implement our fine-grained access control policy in OpenMRS, we first made the `patientId` field of the `Patient` class a role parameter via the `@RoleParam` annotation. We then replaced the original privilege annotations on the `PatientService` interface with `@Requires` annotations. For example, the declaration of the `getPatient` method is now:

```
@Requires(roles={"ProviderFor", "Patient", "Supervisor"},
          params={"patientId", "patientId", ""})
public Patient getPatient(@RoleParam final Integer patientId)
    throws APIException;
```

This method fetches the patient identified by `patientId` from the database. To call it, the caller must possess either the *ProviderFor*, *Patient*, or *Supervisor* roles. These first two roles are parameterized by the specific `patientId`, while *Supervisor* is unparameterized.

To provide dynamic role checks, we first created three new privileges in OpenMRS, corresponding to our three roles: `ORBAC_PATIENT`, `ORBAC_PROVIDER`, and `ORBAC_SUPERVISOR`. Each of these privileges has an associated OpenMRS role, which can then be assigned to users. We added role predicate methods for each of our ORBAC roles to the `Context` class. For example, the role predicate for the *Patient* role is defined as follows:

```
@RolePredicate(roles="Patient", params="patientId")
public static boolean hasPatientRole(
    @RoleParam final Integer patientId) {
    User user = Context.getAuthenticatedUser();
    if (user==null || !Context.hasPrivilege("ORBAC_PATIENT"))
        return false;
    else return user.getUserId().equals(patientId);
}
```

The method checks if the user has the OpenMRS privilege `ORBAC_PATIENT` and if so it compares the user's identifier to the specified patient identifier.

Checking the OpenMRS source code To ensure that the required roles for accessing patients were enforced, we ran our pluggable type system on the entire OpenMRS code base (a total of 633 Java files). The checking takes 11 seconds on a MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor and 2 GB of memory.

We used our type checker in an iterative manner in order to add necessary annotations and dynamic checks until all type errors were resolved. In general we used `@Requires` annotations on methods to remove static type errors. As mentioned in Sect. 2.2, we cannot place a `@Requires` annotation on the top-level methods in servlets through which all user requests must pass. This is the natural place to use predicate methods that perform dynamic security credential checks to satisfy the type checker.

In total, we made changes to 81 (13%) of the files. A total of 298 `@Requires` annotations and 151 dynamic checks were added. Since the pluggable type system successfully checks the code, the dynamic role checks that occur within servlet code are guaranteed to be sufficient on all paths to the protected methods of `PatientServiceImpl`.

The count of dynamic checks represents individual role predicate calls (`hasPatientRole`, `hasProviderRole`, or `hasSupervisorRole`). In many cases, these predicates are used together in a single `if` statement. In general, dynamic checks for patient reads use a disjunction of all three predicates, checks for patient writes use a disjunction of the provider and supervisor predicates, and checks for servlets that generate reports (which access many patients) use the supervisor predicate alone.

4.4 Limitations and tradeoffs

Final fields and role parameters In the ORBAC example of Sect. 2, role parameter fields are declared as `final`. Our type system requires that role parameters do not change. If role parameters can change, the type system becomes unsound, potentially allowing prohibited calls.

Unfortunately, Hibernate requires that persisted objects have default constructors and non-final id fields. These id fields are frequently the same fields used as role parameters (e.g., the `patientId` of class `Patient`). To address this, we permit role parameter fields to be non-final but include checks in our pluggable type system to ensure that role parameter fields are not assigned outside of constructors. We also use the JavaCOP flow framework to ensure that every constructor initializes all role parameter fields.

ProviderFor vs. Provider roles In our case study, we chose to define for doctors a *ProviderFor* role which is parameterized by a patient id. This approach is straightforward and easily handles the case where a patient has multiple providers. However, it is problematic when representing collections. For example, the `getPatientsByName` method of `PatientService` takes a partial patient name and returns a `Collection<Patient>` of matching patients. The names of these patients are then displayed to the user, who can drill down to a specific patient record. We changed this method to return only those patients accessible to the user. Unfortunately, there is no way to represent the precise element type of this collection in our type system, since each patient has a different id. Therefore, we use a collection object with no role parameter. This lack of static validation cannot cause a security violation, but it does necessitate the use of dynamic role predicate checks in order to fetch the actual `Patient` object when the provider "drills down."

An alternative would be to instead use a *Provider* role, which is parameterized by the doctor's user id. Thus, the patients returned by `getPatientsByName` would all be parameterized by the same value, allowing easier representation in our type system.

This alternative approach is not without disadvantages. In the most obvious implementation of this policy, the `Patient` object would be parameterized by two fields: `patientId` and `providerId`. However, this does not work well if a patient can have multiple providers. One work-around is to change the `getPatient` method for `PatientServiceImpl` to populate the `providerId` with the current user's id, if the user is in the set of providers for the patient.

Access control for encounters and observations In our current implementation, accesses to objects logically contained within patients, such as encounters and observations, are not protected by `@Requires` annotations. In theory, this could lead to an

unsoundness in the security policy, although, in practice, the OpenMRS navigation design prevents users from accessing these sub-objects without first accessing the parent `Patient` instance. To be sure there is no violation, we could add `@Requires` annotations to encounters and observations. Alternatively, we could use a form of *object ownership* [7] to verify that these objects are in fact logically contained within their associated patient objects.

5 Related Work

Role-based access control [9] has been used successfully in many systems and is now a NIST standard. Several approaches have been explored by researchers to extend declarative access control models like RBAC to represent and enforce instance-level policies. However, these approaches have employed only dynamic enforcement of such policies.

The emphasis in some prior work [1, 15, 4] is on clarifying the formal semantics of a parameterized access control model. For example, Abdallah and Khayat [1] provide a set-theoretic semantics in a formal specification language, and Barth *et al.* [4] briefly mentions a parameterized role extension to a temporal logic for reasoning about privacy. We adapt a variant of these generalized RBAC models to an object-oriented language, provide a static type system for enforcing access control, and have implemented and validated the approach in Java.

The Resource Access Decision facility (RAD) [3] extends RBAC-based access control policies with access checks based on user relationships. Policies may be configured to require certain relationship predicates to be true when an activated role is used to access an object. For example, a rule might state that doctors can only access the records of patients to which they have an *attending* relationship. However, these relationship predicates are not defined in a declarative manner — a CORBA interface must be implemented in the application to evaluate each predicate. This precludes any use of a static analysis based on the relationships required by a policy.

The database community has also addressed the enforcement of instance-level access control policies (e.g., [12, 26, 20, 22]). In particular, [12] extends RBAC with parameterized *role templates*, where the parameters of a template refer to database columns or constants and serve a similar function as our role parameters. Implementing fine-grained access control policies at the database level has two key advantages: one can define policies directly on the data to be protected and the filtering of records can be integrated with query optimization. However, database-level access control also has several disadvantages. First, it would be very difficult to statically determine the code paths in an application which lead to a given dynamically-generated SQL statement, which would be necessary to statically detect access violations. Second, developers may also want to enforce restrictions on function invocations in the application, which would require a separate mechanism from the database-level access control policies. Third, most modern application deployments store the mapping of users to roles in an external repository (e.g., an LDAP server). Information stored in such a repository might not be available to the database query engine.

Instance-level access control policies can also be defined using domain-specific languages. For example, the XAML standard [8] permits the definition of access policies

for web services which reference data in individual request messages. Cassandra [6, 5] extends Datalog to express constraint rules referencing parameterized RBAC-style roles. These approaches are appropriate for enforcing access control *between* applications but are not so easily applied *within* an application. To (dynamically) enforce such policies within an application, one would need to map the entities referenced by the policy to actual object instances. In addition, the more expressive semantics of these policies would complicate static analysis.

We enforce access control policies through explicit dynamic and static checks added to the codebase through annotations. One could also write policies in a separate language outside the codebase and automatically insert them into the code at compile time or runtime (via bytecode manipulation). This approach has been explored [23], with policies expressed as *access constraint* rules — boolean expressions over an object and its relationships. Our ORBAC annotations could be translated to access constraint rules.

Our approach is orthogonal to *Hierarchical RBAC* [27], where a partial order is defined on roles. If a role R_1 is greater than a role R_2 in this hierarchy, then any user holding R_1 also holds the permissions associated with R_2. This hierarchy is statically defined and not dependent on individual object instances, so it still only supports coarse-grained policies. For example, if a *Physician* role dominates a *Healthcare-Provider* role in the hierarchy, assigning two users to *Physician* roles gives them the exact same permissions, which are a superset of the permissions granted to users assigned the *Healthcare-Provider* role. One could extend our ORBAC model to support hierarchies by including a partial order on (parameterized) roles.

There has also been work on static analysis for RBAC systems. Closest to our work is that of Pistoia *et al.* on static analysis of security policies in Java EE [25]. They employ an interprocedural analysis to identify RBAC policies that are *insufficient* (i.e., can lead to runtime authorization failures), *redundant* (i.e., grants more roles than necessary), and *subversive* (i.e., allows bypassing access control requirements). Our static type system prevents the first and third of these errors, but for the more expressive ORBAC model. Using a type system as opposed to an interprocedural analysis allows us to provide modular guarantees about proper access control on each function in a scalable manner, at the expense of requiring user annotations.

Researchers have explored many forms of dependent type systems [18], whereby types depend on program values. The closest to our work is the notion of *constrained types* in the X10 programming language [19]. In X10, classes are explicitly parameterized by a set of *properties*, which are treated within the class as public final fields. Our design is similar but uses annotations to implicitly parameterize a class by a designated set of fields without modifying Java's syntax. Similarly, an X10 type has the form `C{e}`, where `C` is a class name and `e` is a constraint on the class's properties, while we use annotations to specify constraints. In our type system, these constraints are always simple equality constraints. The X10 compiler has built-in support for checking equality constraints, but it also allows users to plug in solvers for other constraints.

The static checking of roles in our type system has no analogue in X10's constrained types. This part of our type system is most closely related to *type-and-effect* systems [11], which statically track a set of computational effects. The computational effects we track are the privileged operations that a function may invoke, which de-

termine the roles that are allowed to invoke the function. Roles are also similar to *capabilities* [29], which are a dual to effects. However, roles are disjunctive rather than conjunctive: it is sufficient for an execution to hold *any* of a function's roles, while capability systems require all capabilities to be held to ensure proper execution.

6 Conclusions

We have presented the design, implementation, formalization, and practical validation of Object-sensitive RBAC (ORBAC), a generalization of the widely used RBAC model for access control. ORBAC allows different instances of the same class to be distinguished by a designated set of object properties. These properties can then be used to parameterize roles thereby supporting fine-grained access policies that are useful in common scenarios but hard to implement in traditional RBAC. We have implemented a novel static type system that employs forms of dependent types and flow sensitivity to provide sound yet precise reasoning about an application's adherence to an ORBAC policy. Our OpenMRS case study illustrates the practical utility of the ORBAC model and our type system in a realistic setting.

We have focused on a useful but restricted version of ORBAC. This model can be naturally extended to support a more expressive policy language. Our current JavaCop-based implementation could be enhanced to support role predicates as arbitrary propositional formulas as well as multiple parameters per role, both of which are in our formalization. Useful extensions to the type system presented here include the addition of a partial order on roles, a richer constraint language for index values, and static tracking of the temporal order of privileged operations. Finally, we would like to investigate both local and global type inference of object-sensitive roles.

References

1. A. E. Abdallah and E. J. Khayat. A formal model for parameterized role-based access control. In T. Dimitrakos and F. Martinelli, editors, *Formal Aspects in Security and Trust*, pages 233–246. Springer, 2004.
2. C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 57–74. ACM Press, 2006.
3. J. Barkley, K. Beznosov, and J. Uppal. Supporting relationships in access control using role based access control. In *RBAC '99: Proceedings of the fourth ACM workshop on Role-based access control*, pages 55–65. ACM, 1999.
4. A. Barth, J. Mitchell, A. Datta, and S. Sundaram. Privacy and utility in business processes. In *CSF'07*, pages 279–294. IEEE Computer Society, 2007.
5. M. Becker. Information governance in nhs's npfit: A case for policy specification. *International Journal of Medical Informatics (IJMI)*, 76(5-6), 2007.
6. M. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *POLICY '04*, pages 159–168, 2004.
7. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64. ACM Press, 1998.

8. eXtensible Access Control Markup Language (XACML) Version 2.03. OASIS Standard, February 2005.
9. D. Ferraiolo and R. Kuhn. Role-based access control. In *15th National Computer Security Conference*, 1992.
10. J. Fischer, D. Marino, R. Majumdar, and T. Millstein. Fine-grained access control with object-sensitive roles. Technical Report CSD-TR-090010, UCLA Comp Sci Dept, 2009.
11. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 28–38. ACM Press, 1986.
12. L. Giuri and P. Iglio. Role templates for content-based access control. In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 153–159. ACM, 1997.
13. Hibernate home page. `http://www.hibernate.org`.
14. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
15. T. Jaeger, T. Michailidis, and R. Rada. Access control in a virtual university. In *WETICE '99: Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, pages 135–140, Washington, DC, USA, 1999. IEEE Computer Society.
16. Java Platform, Enterprise Edition home page. `http://java.sun.com/javaee`.
17. S. Markstrum, D. Marino, M. Esquivel, and T. Millstein. Practical enforcement and testing of pluggable type systems. Technical Report CSD-TR-080013, UCLA Comp Sci Dept, 2008.
18. P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.
19. N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 457–474, New York, NY, USA, 2008. ACM.
20. L. Olson, C. Gunter, and P. Madhusudan. A formal framework for reflective database access control policies. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 289–298. ACM, 2008.
21. OpenMRS home page. `http://openmrs.org`.
22. Oracle 11^g Virtual Private Database, 2009. http://www.oracle.com/technology/deploy/security/database-security/virtual-private-database/index.html.
23. R. Pandey and B. Hashii. Providing fine-grained access control for Java programs. In *ECOOP '99*, volume LNCS 1628/1999, pages 668–692. Springer, 1999.
24. B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002.
25. M. Pistoia, S. Fink, R. Flynn, and E. Yahav. When role models have flaws: Static validation of enterprise security policies. In *ICSE '07*, pages 478–488. IEEE, 2007.
26. S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 551–562. ACM, 2004.
27. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
28. Spring Application Framework home page. `http://www.springsource.org`.
29. D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.*, 22(4):701–771, 2000.
30. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994.