# Coordinating Accessibility versus Restrictions in Distributed Object Systems

Arnon Rosenthal
The MITRE Corporation
202 Burlington Road, k/308
Bedford MA, 01730-1420 USA
arnie@mitre.org +1- 781-271-7577

Chris Clifton
Purdue University
1398 Computer Science Building
West Lafayette, IN, 47907-1398 USA
clifton@cs.purdue.edu +1-765-494-6010

**Abstract**

*This work aims to provide administrators with services for managing permissions in a distributed object system, by connecting business-level tasks to access controls on low level functions. Specifically, the techniques connect abilities (to complete externally-invoked functions) to the access controls on individual functions, across all servers. Our main results are the problem formalization, plus algorithms to synthesize "least privilege" permissions for a given set of desired abilities. Desirable extensions and numerous research issues are identified.*

*Keywords: Access controls, distributed object management, security, business tasks*

## 1 The Problem

We believe distributed systems' security faces a grand challenge – *to make security administration so easy that ordinary organizations will do it well.* To make it easy, enterprises will need automated help--a security administrator's assistant for distributed object computing (*d. o. c.*). It would aid analysis and synthesis of permissions, based on an understanding of how access controls on function invocation connect to abilities to invoke the functions needed to complete a work task. This paper begins a theory on which to base such assistants

### 1.1 Goals

Security policy is typically based on specifying what needs to be protected – a model of protecting against bad access. We see an alternative approach – a model of permitting only needed accesses. Specifying what permissions are required to complete a task, as opposed to what accesses violate policy, has the potential to provide tighter controls without restricting useful work.

We want to provide services that help administrators to coordinate accessibility versus protection in a distributed object system, e.g., the one illustrated in Figure 1. Such systems can be seen as collections of diverse types of servers – object middleware, database managers, and specialized application systems. Each server hosts one or more interfaces, consisting of functions that can be invoked. The code that implements the function may invoke other functions, in the same or other servers. We consider only requests made through these server-controlled interfaces – we do not look "under the covers" to see other information flows among programs running within a server.

The need for *abilities* to accomplish work is modeled as running functions to completion, unhindered by access controls. The process of balancing task accessibility versus resource protection is particularly difficult, because ability needs span many invocations, and access controls deal with single ones. Distributed object systems introduce further difficulties, with different server characteristics and spans of control.

We first discuss some strategies for automated synthesis, based on the execution model and the "principle of least privilege". Our top-down synthesis takes business tasks' desired abilities as a constraint. It then seeks to impose the tightest feasible access controls on each function consistent with 1) those abilities and 2) the capabilities of each server. We are currently working on the complementary (bottom-up) analysis theory, to determine the abilities that stem from an arbitrary set of access permissions.

### 1.2 The Services to Be Provided

Our work provides models that automatically maintain connections between the following:

- *Servers' access control policies* – predicates that limit the incoming requests that a server will invoke (i.e., begin to execute). Functions and data that are subject to such limitations are called *protected* by this access control system.
- *Abilities to complete work:* These describe the ability to complete a function that represents the automated processing needed for some business task. Completion requires that each onward call satisfy the access policy of the servers that execute it.

Administrators will supply ability predicates that answer questions like the following:

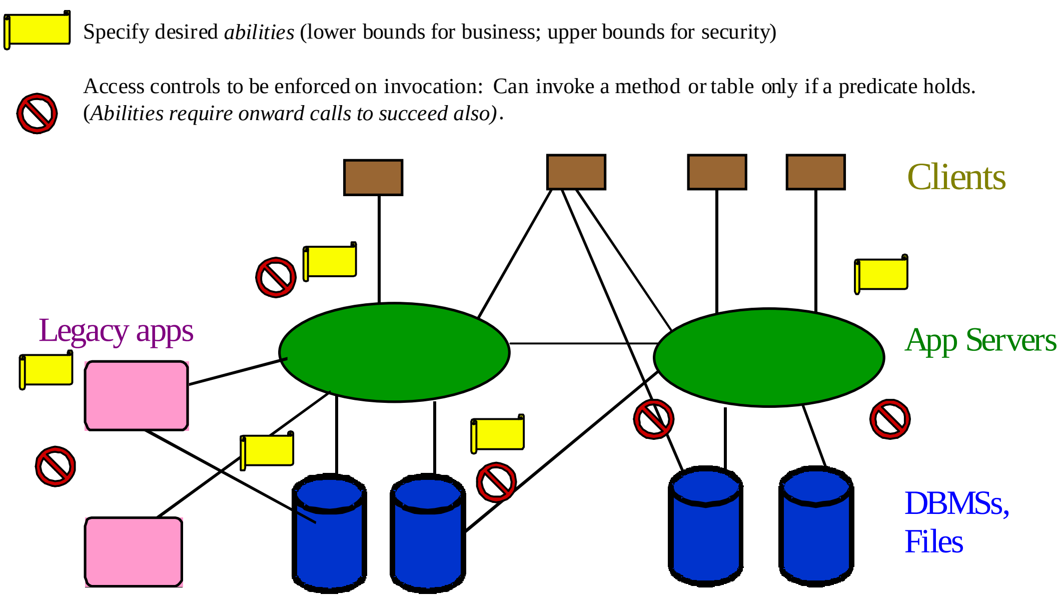
Specify desired *abilities* (lower bounds for business; upper bounds for security)
Access controls to be enforced on invocation: Can invoke a method or table only if a predicate holds. (*Abilities require onward calls to succeed also)*.
Clients
Legacy apps
App Servers
DBMSs, Files

**Figure 1: A Typical Distributed Object System: Many Interfaces, Many Stakeholders**
**Our goal for access control: Specify where convenient. Enforce where convenient.**

- Who can hire an employee into the Engineering department?
- Who can mark financial software as 'tested'?
- Who can update Accounts Payable with amounts over $1000?
- Who can discharge which patients, for what reasons (i.e., can run Discharge(Patient_Id, Date, Reason, Bill_Id, DoctorName) )?
- Who can issue a particular SQL request to the database information about Boston employees?

Ability predicates will typically be defined over externally-visible request arguments, request context (e.g., user, time issued), and database contents. Access control predicates reference these, and might also reference the (internal) call stack. For example, the database server might grant access to EMP and DEPT to certain users only for requests made in the course of executing the function HireEmployee(Name, Dept, Salary) with Salary<$100,000. Much attention has (rightly) gone into delegation and credentials. To us, these are implementation mechanisms. We are concerned with defining the policies that the mechanisms will implement.

Figure 1 also shows a system's security specifications. For various functions within the servers, administrators specify required abilities (giving lower and perhaps upper bounds); they also may specify upper bounds on the permissions that may be granted. Our theory aims to support tools that would specify the desired access controls on invocations – of a form the function's server can enforce, and such that abilities and controls respect the appropriate bounds.

A policy specified in terms of one interface may be enforced by access controls on other interfaces. For example, a policy that allows Compensation Analysts to run the "Analyze Salaries" application on Permanent Employees could potentially be implemented in at least three ways. One could enforce the policy only on the call to AnalyzeSalaries; one could have the database restrict access to views of the underlying data; or one could enforce in both places.

We impose no restrictions on how an organization distributes the authority to make policies. Administration tools should be able to handle several modes: one global administrator; one administrator for each server; or one administrator for all functions in each business area (e.g., Finance), regardless of server boundaries. For controls over the act of delegation, see [6, 5, 2]. Run-time authority delegation, e.g., by passing credentials, are implementation details hidden below our model.

## 1.3 What Do Systems Do Now?

Today's multi-tier systems do enforce security, but in ways that are far from adequate. None of them (to our knowledge) has tools that can determine what abilities would stem from access restrictions on all the servers involved in a user transaction. Instead, one often sees reliance on just one of the servers. Or else, one typically sees separate administration within each server, and no effective means of coordinating their policies. Below, we describe two common practices today, and explain why they are often not satisfactory.

**1.3.1 Middleware Provides the Only Controls.** In many systems, the database validates that a request is coming from the middleware, but imposes no restrictions on what that request may do. The middleware security may be basic (e.g., predicates just on the user and the function name) or may include a rule engine that provides a powerful language for expressing access controls [3]. These engines can be close to the original point of invocation, and can be replicated relatively cheaply.

A prerequisite for this approach is that other resource owners be willing to trust the middleware; it is less likely to suit architectures where middleware spans several enterprises. Even within an enterprise, this approach has serious limitations, which make it rather low assurance. Failure modes include:

- *Lazy administration*: The middleware policy maker may be careless about the database's resources.
- *Unanticipated behaviors.* Administrators may misjudge the accesses a function may make. The cause may be rare events (e.g., particular input errors), or Trojan horses inserted maliciously.
- *Spoofing*: Someone may tamper with the approved request en route to the database. (Many techniques can reduce this threat, but some risk remains.)

If one generates (nearly) the least permissions that allow desired functions to complete, one might give each function access to three tables rather than a hundred. Thus, one can automatically block many illegitimate accesses created by spoofs of the request, by Trojan horses inserted after permissions were generated, or human error. If a database administrator manually reviews the "least" permissions, many of the remaining errors can be caught.

**1.3.2 Database Provides the Only Controls.** Another typical pattern is to rely on database access controls. Testing values of stored data can be easier and more efficient in the DBMS. Also, data owners can enforce their own restrictions locally, if they do not trust the middleware, or they wish to continue using controls that are already specified. But giving the DBMS full responsibility has three categories of disadvantages:

- One sometimes wants to give extra privileges to trusted functions. Business functions increasingly run in the middleware, but today's DBMSs recognize trusted functions only if they run in the database (i.e., database view or stored procedures). Fortunately, the credentials/PKI features being added to DBMSs for authenticating users also will apply to authenticating functions [1].
- Enforcement at the DBMS is not on the request the client generated, but rather on a request descended from the client method. This distance makes it harder to frame an understandable error message.
- One wants to push processing to stateless middleware objects, rather than to the database. As workload increases, one can easily add processors that run copies of stateless services. A database is not so cheaply replicated, because one must constantly ship updates among the copies.

An automated assistant could inform application administrators which functions appear to have enough permissions to complete. It can also help a data administrator include the identity of the calling function in permission predicates.

For all these reasons, it may be preferable to have multiple lines of defense. Instead of "give free rein to middleware-approved functions" enforcement only against the *data* model, we want the tightest permissions that 1) allow function executions that implement the approved abilities, and 2) that the DBMS server can enforce.

## 1.4 Scope Limitations

Basic limitations are mentioned here; others appear as Future Work. First, it can be impossible to predict a function's behavior—the general problem is Turing complete. We focus, though, on tractable cases that administrators understand and routinely handle. Second, we treat the system as a *static* collection of interfaces, functions, and access control predicates. We do not address performance and assurance of the enforcement strategies.

We do not consider application failure (e.g., if the application voluntarily decides to signal failure, or its code crashes) to be a security administration problem For us, ability is taken just as "adequate access privileges".

Despite the limitations, the work still is general. We make no assumptions about the implementation language, about argument-passing mechanisms, or if function calls must return before the caller resumes operation. The model does not constrain concurrency or isolation; it tacitly places all isolation and coordination logic (e.g., private versus shared storage, transaction management) as part of the function's semantics.

## 1.5 Contributions of this Work

Our aim is to open up the multifaceted problem, to allow productive work by specialists in each facet. Security policy experts might describe the forms of predicates that are most needed for interesting policies. Experts in code and data flow analysis might find predicates that relate the states of a function and its onward calls. Datalog experts might provide techniques for handling graphs with cycles [8]. Experts in middleware and DBMSs' security services can implement additional predicate types, e.g., on the call history. And security management experts can build tools that exploit synthesis and analysis to simplify administration.

In Section 1 we motivated the problem. Section 2 formalizes the Execution model, which describes how requests execute as part of a distributed object computing system. Together, these sections extract the problem from the morass of real world issues, and describe the services desired.

Section 3 provides algorithms to synthesize "minimal" permissions that will guarantee the desired abilities. For now, we synthesize based on the "principle of least privilege". That is, we impose the tightest access permission predicates from which we can infer that the desired abilities will be present. We state open problems (conjectures) about whether the synthesis's bounds are tight. The final section sketches an agenda for the necessary additional research.

## 2 Execution Model

Concurrent, heterogeneous systems are notoriously difficult to model. Yet humans and tools can often predict (or at least, bound) some behaviors. The execution model describes perfect knowledge. It gives a basis for reasoning later using bounded knowledge.

We base our model on a few key ideas:

- Access is controlled at the function level. The granularity of a function determines how fine-grained the access controls are (e.g., each access to a data item could be an independent function call to model data item access controls.) A function may call other functions.
- A function call happens in the context of a system state. The state captures everything relevant to access controls.
- Each function has an access predicate that determines (based on system state) if the call is permitted.

Composing these ideas allows us to model the access permissions (abilities) needed for a function to complete.

### 2.1 The Basic Models: Machine and Function Call

The entire system (including its input stream) is seen as a deterministic state machine. The system *state* is the product of the states of all the memory locations, including ordinary memory, *control memory* (e.g., call stacks, return codes, scheduling information), *security memory* (e.g., access predicates, user credentials), and the sequence of future input values. The machine executes sequentially and deterministically, consuming one input (usually *null*) at each step. State and function behavior contain (for theoretical reasoning) everything needed to simulate future execution of the system. (Tools will reason with predicates that describe bounds.)

The function layer describes execution as calls to functions described in the server interfaces. *Callstates* are states of the machine that correspond to a function call; a callstate S that invokes function f is denoted $S_f$ . The model shows callstates, access predicate evaluation, function execution, and function return states. We are particularly interested in the new callstates generated during call execution (called *onward callstates*), and in whether the access controller allows those invocations. Aside from these, we abstract away the execution logic.

The semantics are: When the function f is invoked, its access predicate is evaluated *using data in the callstate*. (We do not specify how the access control predicate is implemented – Access Control Lists, Capabilities [7], or Security Meta Objects [4] can be used.) If the result is *true* (for that callstate), the server creates a *call execution* and begins executing the function; if *false*, the next (and only) step from this call is the return event with return code "failed". Useful work is assumed to correspond to completion of a call execution.

Access predicate evaluation is treated as part of the call state execution; with no side effects on the system state. An extended model might allow actions to write to the log, or to attach information to the user's request, as in [3]. Steps from different active executions can interleave, scheduled by mechanisms outside our model.

A function execution trace is illustrated below. Ordinary operations (denoted *o*) are not relevant to access predicates or the callstack. Execution begins with a *system* call to function $f_1$ (call execution $C_a$). After two ordinary operations, a call from *system* to $f_2$ begins call execution $C_b$, that then invokes $f_3$ (execution $C_d$). $C_b$ continues executing and interleaves its calls with callee $C_d$. The code of the executing functions determines (below the level of our model) what the next state shall be, and what function's execution occurs at that step.

| *Call-Execution this step belongs to* | *Operation Executed* | *Identifier of Call- Execution created by This Step* |
|---|---|---|
| [sys] | $f_1$ | <$C_a$> |
| [$C_a$] | o | |
| [sys] | o | |
| [sys] | $f_2$ | <$C_b$> |
| [$C_b$] | $f_3$ | <$C_d$> |
| [$C_d$] | o | |
| [$C_a$] | return | |
| [$C_b$] | o | /* model lets execution interleave with $C_d$ */ |
| [$C_d$] | o | |
| [$C_d$] | return | |
| [$C_b$] | return | |

Our algorithms traverse a function *call graph*, which has a node for each function, and an edge from f to g if f can call g. This paper considers only acyclic graphs. Nodes with no input edges will be called *sources*, and nodes without outputs will be called *sinks*. If one works with a superset of the call graph, one will grant greater permissions, and still guarantee the needed abilities.

The next definition is crucial: Given a callstate $S_f$ we define its *onward callset* (or just *callset($S_f$)*) to be the set of call states associated with the execution of f, when f is invoked in $S_f$ .

We note several important properties:

- The callset includes only direct calls from the code of f, not calls from f's callees. One obtains the callset by understanding the code of f, not the callees.
- $S_f$ contains *all* information relevant to the execution of f.
- The callstates in the onward *callset($S_f$)* are equally well defined, so behavior can be examined transitively.

The administration process confronts function code in just one place – to describe (or bound) the onward callset.[1] The means of obtaining such bounds are outside the model. Sophisticated bounds are left for experts who understand the function code, and the mechanisms (private storage, transactions) used to separate function executions.

## 2.2 The Model for Deriving Abilities

We focus on abilities, and their derivation from permissions. The key question is "Do the access controls permit all onward calls associated with this call's execution?" This leads us to the basic rule, which will drive the rest of the paper.

*Basic Rule:* You have the ability to complete f if and only if you have the permission to invoke f and the ability to complete everything directly called by f..

From the verbal statement, we get a formal recursion, a Boolean equality for each callstate.

$$ability(S) = accessPredicate(S) \wedge (\wedge\{\text{ability}(X^j) \mid X^j \text{ in } callset(S)\}) \qquad (1)$$

This formula can be interpreted as describing the ability to complete from a particular state *S.* But a more interesting interpretation is that it defines a predicate on callstates. Next, we add the externally specified ability requirements that drive synthesis. Let externallyDesiredAbility(S) denote a predicate that tells the minimum ability desired for each callstate.[2] Then

$$ability(S) \geq externallyDesiredAbility(S) \qquad (2)$$

Theorem: Expressions 1-2 have a unique least fixed point (which we call their *solution.*)

*Proof Sketch:* The sequence constructed by the following algorithm converges to the fixed point. We assume the state set is finite.

```
For each state S, initialize
    $ability(S) := externallyDesiredAbility(S).$
Iterate until no change to any state's ability {
    $AccessPredicate(S) = ability(S)$
    For each pair of states (S, $X^j$) such that is $X^j$ is in
        callset(S)
    $ability(S) := ability(S) \vee ability(X^j)$ }
```

For each S, ability(S) increases monotonically on each iteration, and has an upper bound (*true*, or alternatively, the disjunction of all externallyDesiredAbility predicates). Hence it converges to an upper bound ability*, which can be shown to be a fixed point. To show uniqueness, we intersect all predicates that are fixed points. This can be shown to be a fixed point, and is the unique least one. QED

Similar treatments are possible if one imposes *upper* bounds, either on what invocations will be allowed (upper bounds on *AccessPredicate*) or on what callstates might be able to complete (upper bounds on *ability*). Now there is a greatest fixed point, showing the greatest abilities consistent with the protections. One might also mix both kinds of upper bounds. A further extension would be to allow mixes of both upper and lower bounds; in such situations, administrators will need help in detecting and resolving inconsistencies.

We now know how to analyze any single state, by traversing the call graph onward from that state. But no administrator cannot examine every state, nor write predicates that predict all functions' behavior. The next section chooses new predicates to describe collections of states, and uses whatever bounds on callsets are available. To avoid being mired in complex algorithms, we handle only the acyclic case.

## 3 Synthesizing Access Permissions to Provide Abilities

We now exploit the Basic Rule to synthesize access permissions based on the *principle of least privilege.* For each computational step, we grant the least access permissions (tightest predicates), for which we can demonstrate that externally-desired call executions will complete. For example, we do not give blanket privileges to all middleware-approved function calls; we would give permission only on the database objects that the functions ought to access, and only for calls onward from the approved middleware functions.

The subsections below synthesize permission predicates of different sorts, to meet the goals above. Section 3.1 presents the general algorithm, and Section 3.2 discusses the simplest special case. Section 3.3 allows more information about functions' behavior. For servers that offer a limited predicate language, Section 3.4

[1] Callset bounds are useful for purposes beyond security. For example, one may wish to load a mobile platform with all the resources that will be accessed by critical services running on it.

[2] To avoid self-referential definitions, we assume that externallyDesiredAbility and accessPermission predicates do not reference {access predicates}. A weaker condition, monotonicity, would suffice.

produces predicates that fit within each server's capabilities.

## 3.1 Synthesizing Permissions to Guarantee Abilities

Some conventions and notation will be needed before the algorithm can be presented. We treat a predicate as synonymous with a set of states, using whichever form is locally more convenient. Let g denote function; *edap*$_g$ will denote its externally desired ability predicate. Suppose that there exists a callstate $S_g$ satisfying *edap*$_g$ such that g or one of its callees generates a callstate $S_f$ for f – that is a call to f is a descendent of the call $S_g$. Then we say f (and more specifically, $S_f$) is *needed for* g[*edap*$_g$].

$p_f$ will denote the desired ability predicate for f, combining externally desired ability for f to complete, and ability due to ancestors g that need f That is, $p_f$ is defined as $\{S_f | S_f \varepsilon \text{ edap}_f$ or $S_f$ is needed for some $g[\text{edap}_g]\}$. Looking onward from f, define the *call-mapping function* $cm(S_f)$ to yield {all callstates invoked by the execution from $S_f$}. If the call graph shows that f *might* call $f_i$, $cm_{<f, f_i>}(S_f)$ denotes {invoked callstates that invoke $f_i$}. The call mapping function for a set C of callstates for f is defined in the obvious way, as $\cup\{cm(S_f) \mid S_f \varepsilon C\}$.

The synthesis algorithm lets desired abilities propagate from each f. The algorithm exploits the intuition of equation (1), that the ability to complete from $S_f$ requires the ability to complete from each callstate generated during its execution. The call mapping function may not be known in a tractable form that we can apply to the desired set of states. Fortunately, we can use any predicate that contains the resulting calls to each successor $S_{fi}$, and we will get abilities that contain the desired ones.

We focus on predicates of a convenient form that often lets us make progress. Recall that the callstack is part of the state. Thus, by definition, for each $f_i \varepsilon$callstates(f), the predicate (f is parent($f_i$) in callstack) holds. We can thus seek upper bound predicates of the form (f is parent($f_i$) in callstack) $\wedge$ (any other upper bound). We call the second conjunct the *specific propagated desires*, denoted *sprop.*

Pragmatically, we want *sprop*(f, $f_i$) to be as tight as possible, to be simple to manipulate in larger expressions, and to be enforceable by servers. *True* is always a legal choice, if we cannot infer a tighter bound. Later subsections will explore situations where more helpful upper bounds can be inferred for *sprop*. Servers' limited enforcement abilities are considered in the last step, after all desired predicates have been calculated.

The synthesis algorithm traverses the call graph from sources to sinks, always respecting the calling order. Administrators express application requirements by specifying a lower bound predicate, denoted *externallyDesiredAbility*(f) or edap$_f$, for ability to complete each function in the call graph. The (total) desired ability to complete $f_i$ is computed by OR-ing its externallyDesiredAbility with the needs propagated from all its parents.

The algorithm below will derive a set of access predicates. When installed, these predicates drive enforcement. For any such set *accPreds*, let *abilities*(accPreds)* denote the resulting abilities, i.e., the resulting least fixed point of equation 1.

<u>*Simplified Synthesis Algorithm:*  Determine sufficient permissions</u>

```
In   externallyDesiredAbility(f): An ability predicate for
        each function f
Out  accessPermissions(f): A permission predicate for
        each function f, such that the system with these
        permissions will exhibit all the externally desired
        abilities.
Out  desiredAbility(f): The ability predicate obtained with
        the above permissions.

Postcondition: For all f,  the access predicates are set
        such that the ability to complete f ≥
        externallyDesiredAbility(f)

For each function x                    /* Initialize each node*/
        desiredAbility(x) = externallyDesiredAbility(x)

/* traverse top down, from sources to sinks */
Visit each non-source node f in graph order
/* Create permissions so the accumulated desired calls
        can be invoked */
accessPermission(f) = desiredAbility(f)

For each $f_i$ in fnCallset(f) /* $f_i$ permissions must allow f to
        complete */
            /* propagate the abilities that the parent requires */
Determine a specific propagated predicate, denoted
        sprop(f, $f_i$)
desiredAbility($f_i$) = [(f is parent($f_i$) in callstack) and
        sprop(f, $f_i$)] OR desiredAbility($f_i$)

For each f
Round up accessPermission(f) to a predicate that the
        server for f can enforce.
```

***<u>Main Theorem</u>*** The above algorithm yields sufficient access permissions. That is, for each f, externallyDesiredAbility(f) $\subseteq$ abilities*(accPreds)

*Proof*:

It will be sufficient to prove (by induction) two hypotheses about the algorithms' results:

- *Invocation:* For each "needed" call, we have permission to *invoke* the call.

- *Completion:* For each "needed" call, we can *complete* f.

*Proof of Invocation hypothesis:* To see that invocations will succeed, perform induction top down (from calls to source nodes).

For every node, the DesiredAbility is initialized to its own externally desired ability, and changed only to make it less restrictive, by OR-ing in additional predicates[3]. Source nodes receive only external calls, so the initial permissions suffice. To complete the induction, we now prove that the access predicate will allow invocations needed for non-source nodes to complete.

Consider a non-source node $f_i$ that, for some predecessor v, is needed for $v[edap_v]$. Let f denote $f_i$'s immediate predecessor on a call path from v. f precedes $f_i$ in the traversal, so (by inductive hypothesis) its access predicate allows f to be invoked on any state needed for $v[p_v]$. When the algorithm traversed the edge from f to $f_i$, the term OR'd into desiredAbility (and from there, to accessPermission) accepted all callstates of $f_i$ reached from the desired states of f. QED

*Proof of Completion hypothesis:* To see that execution will succeed in completing, we now do induction in the reverse direction. First consider sink nodes, which have no onward calls. The previous induction proved that they could indeed be invoked, on any needed calls. Since they have no onward calls, they can complete. The base case holds.

Now consider a needed invocation for a non-sink node f. By the invocation hypothesis established above, f can be invoked. By inductive hypothesis, each of its callees is later in the graph, and hence can complete. Thus, the call to f is able to complete. **QED**

**Discussion:** The algorithm is based on determining callstates and, based on them, choosing an upper bound *sprop*. The conjectures below state first, that tightening the analysis helps reduce the unnecessary abilities and access permissions, and second, that the algorithm does as well as is possible, based on the analysis.

We say that the code $analysis_1$ is *tighter* than $analysis_2$ if callsets($analysis_1$) $\subseteq$ callsets($analysis_2$) and *sprop*($analysis_1$) $\subseteq$ *sprop*($analysis_2$). *Let accPreds($analysis_i$) and abilities*(accPreds($analysis_i$))* denote the results of the algorithm using $analysis_i$.

<u>Conjectured Theorem</u>: If $analysis_1$ is tighter than $analysis_2$, the system is at least as secure. That is, for all f, accessPredicates($analysis_1$) $\subseteq$ accessPredicates($analysis_2$) and abilities*(accPreds($analysis_1$)) $\subseteq$ abilities*(accPreds($analysis_2$)).

<u>*Conjectured Theorem:*</u> Assuming that the analysis results are the tightest possible, then no tighter set of access predicates can be found. (That is, for any tighter set of access predicates, there are functions with the indicated callsets whose needs would not be met.)

[3] Since externallyDesiredAbility predicates do not reference descriptions of access predicates, establishing an access permission does not cause other predicates suddenly to fail.

## 3.2 A Simple Tractable Case

We say a desired ability predicate p is *preserved(f, $f_i$)* if whenever f calls $f_i$, $p(S_f) \Rightarrow p(cm(S_f))$. That is, the steps between callstate(f) and the step invoking $f_i$ do not reduce the truth of p. Predicates such as user identity, and time of day are typically preserved.

When desiredAbility(f) is preserved, then set *sprop*($f_i$) to desiredAbility(f).

<u>Observation</u>: If desiredAbility is preserved(f, $f_i$), then *sprop* has indeed been assigned an upper bound, i.e., exactPropagatedDesires(f, $f_i$) $\subseteq$ *sprop* = desiredAbility(f)

The next theorem, proved by a simple induction, is helpful in verifying that more complex desiredAbility expressions are preserved.

<u>*Lemma:*</u> Suppose each of $p_1, \ldots, p_k$ is (f, $f_i$)-preserved. Let $B(x_1, \ldots x_k)$ be a nonnegative Boolean expression (i.e., using just $\wedge$ and $\vee$). Then $B(p_1, \ldots, p_k)$ is (f, $f_i$)-preserved.

## 3.3 Synthesis Ideas for More Difficult Cases

This section identifies some properties that are less restrictive than preserving all desired abilities, but still justify usefully restrictive *sprop* predicates. Proofs are straightforward, and omitted. We suspect that we have just scratched the surface of exploitable cases.

***Exploiting Conjuncts.*** Suppose we can express the desired ability predicate $p_f = p^{easy} \wedge p^{hard}$, where the first conjunct is preserved(f, $f_i$). We can then set *sprop*= $p^{easy}$.

***Exploiting Disjuncts.*** Suppose callstates(f) can be partitioned as $q \vee q'$ where both q and q' are preserved(f, $f_i$). Suppose that $p_f$ is preserved(f, $f_i$) on executions from states satisfying q. Then we can set *sprop* $= (q \wedge p_f) \vee q'$.

*Motivating Example:* Let ApproveCredit denote the function below. Suppose the desiredAbility predicate $p_{ApproveCredit}$ is = (user is creditAnalyst and customerDesirability < 17). The predicate q is "Amount ≤ 1000"; q' is "Amount > 1000"; both are preserved by ApproveCredit. Note that $p_{ApproveCredit}$ is preserved when Amount <= 1000, so *sprop* $= (q \wedge p_{ApproveCredit}) \vee q'$.

```
Function ApproveCredit(Customer, Amount,
        customerDesirability)
If Amount > 1000 then do
        unanalyzable_update_to_customerDesirability
CreditDecision(Customer, customerDesirability)
```

Then sprop propagated to CreditDecision is set to $((p_{ApproveCredit} \wedge Amount \leq 1000) \vee Amount > 1000)$.

***Exploiting Easy, Localized Transformations:*** Suppose that preservation does not hold, but we understand how $cm_{<f,fi>}$ alters the portion of memory referenced in $p_f$. Some motivating examples are:

- Replace a name by a code, and use it as an argument to a subsequent call. (For example, Massachusetts becomes MA, as a 1:1 function).
- Convert arguments to UPPERCASE and use the converted form as an argument to a subsequent call. (a many:1 function).
- A Name is mapped to one of several social security numbers, based on factors that we cannot predict (a 1:many relation).
- A value is held constant (the trivial case).

Specifically, we require that the call mapping from f to each child $f_i$ be invertable, and that $cm^{-1}$ be known. To handle such transformations, we set *sprop* to be: (some member of $cm^{-1}(S_{fi})$ satisfies desiredAbility(f)

*Proof:* Suppose $S_f$ satisfies desiredAbility(f) and $S_{fi}$ is in $cm(S_f)$. Then by definition, $S_f \varepsilon\, cm^{-1}(S_{fi})$. Hence the disjunct is satisfied for calls from f to $f_i$. QED

We conjecture that the technique can be extended further, e.g., to conditional mappings.

### 3.4 Adapting Synthesis to the Servers' Limitations

We now recall our high level picture that approximates how distributed object systems (DBMSs, webservers, object servers, …) are often organized.

Sometimes we cannot directly impose the desired predicates. Recall that functions are grouped into *interfaces*; and each interface resides in some *server.* A request can be sent from any running process inside or outside the protected system. Several difficulties may prevent us from executing the desired access predicate at the function's server. A server might support too limited a language for expressing predicates, or might refuse to make the callstack available. There may not be an adequate mechanism to pass credentials from the caller's server to the callees. Finally, due to efficiency or security concerns, one may not be able to make remote calls needed to execute the predicates (e.g., to get data).

When it is not possible for a server to enforce the desired permissions, then one selects some predicate enforceable by the server. That is, one "rounds up". Rounding up must satisfy the condition: *If one rounds up the permissions produced by the synthesis algorithm, abilities do not decrease, and hence still suffice.*

## 4 Summary and Conclusions

Permission administration will become increasingly important, as organizations deploy multi-tier and peer-to-peer distributed systems. Metaphorically, the desired control requires gauges that provide information, knobs to turn, and intelligence to choose settings for the knobs. Unfortunately, the gauges and knobs in such systems will soon outstrip administrators' capacity to use them well. Our research aims to give them (semi)-automated tools that increase this capacity.

The area seems theoretically rich, with many open issues. These include:

- *Mechanisms for change.* Since many distributed object systems are 24 x 7, changes will occur even while functions run.
- *Fault tolerance.* I.e., a function's ability to do its job despite failure of some of its onward calls. One could get better bounds if one understood which exceptions could be tolerated.
- *Upper bounds* that guarantee that a resource cannot be too widely accessed. We believe they can be obtained by reversing the direction of bounds used in guaranteeing abilities.
- *Enforcement strategies*: performance and assurance (e.g., trust relationships).
- *Optimality theorems:* Formalize the sense in which the algorithm produces the least privileges, based on desired abilities, server enforcement capabilities, and knowledge of function behavior.

Market demand lies in the future, but there will be several years' lead-time in developing theory and then tools. It seems the right time to begin building a research base so the tools can be principled and powerful.

## 5 References

[1] Oracle 8i DBMS Reference Manual, http://technet.oracle.com/docs/products/oracle8i/doc_index.htm
[2] H. Gladney, "Access Control for Large Collections", *ACM Trans. Information Systems* 15(2), April 1997, pp. 154-194.
[3] Netegrity Site Minder http://www.netegrity.com/
[4] T. Riechmann and F. J. Hauck "Meta Objects for Access Control: Extending Capability-Based Security" *New Security Paradigms Workshop 97*, Great Langdale, UK, Feb. 1998, pp. 17-22
[5] A. Rosenthal, E. Sciore, "View Security as the Basis for Data Warehouse Security", CAiSE Workshop on Design and Management of Data Warehouses, Stockholm, 2000. Also available at http://www.mitre.org/resources/centers/it/staffpages/arnie/
[6] R. Sandhu, V. Bhamidipati, Q. Munawer, "The ARBAC97 Model for Role-Based Administration of Roles", *ACM Trans. Information and System Security,* 2(1), Feb. 1999, p 105-135.
[7] J. S. Shapiro, Jonathan M. Smith, and David J. Farber "EROS: A Fast Capability System", Proc. *17th ACM Symposium on Operating System Principles*, pages 170-185, Kiawah Island Resort, Charleston, SC, Dec. 1999.
[8] J. Ullman, *Principles of Database and Knowledge-Base Systems, vol 1,* Computer Science Press, Rockville Md.