

SPiCE: Configuration Synthesis for Policy Enforcement*

McAfee Research Technical Report #04-016

Deborah Shands, Erik Wu, Jay Jacobs, and James Horning
McAfee Research
{dshands, ewu, jjacobs, jhorning}@nai.com

Stephen Weeks
sweeks@sweeks.com

June 25, 2004

Abstract

Configuring security mechanisms to enforce a high-level security policy is currently a manual, ad-hoc and error-prone process. When access control mechanisms such as firewalls, web servers, databases, file systems, etc. are inconsistently configured, significant security vulnerabilities may result. The problem is especially severe in coalition environments with cross-organizational resource sharing, limited trust and a broad user base. We describe the SPiCE translation system, which simplifies the process of generating configurations for access policy enforcers by providing two tools: (1) a policy editor for writing and refining enterprise-level access control policies for resource sharing in coalition environments; and (2) a multi-target compiler for translating these abstract access control policies into configurations for heterogeneous access policy enforcers. We define the formal semantics of the source and target access control systems and prove the safety and completeness of our translation algorithms. Our results document the characteristics of access control systems that can be modeled in our semantic framework and for which our translation approach is applicable. These results ensure that our work is applicable beyond the access control systems implemented in our prototype policy translation system.

Approved for Public Release, Distribution Unlimited

*Final Report of the SPiCE project, supported by DARPA through the Space and Naval Warfare Systems Center — San Diego under Contract No. N66001-02-C-6021.

1 Introduction

Configuring a suite of heterogeneous mechanisms to enforce a high-level security policy is currently a manual, ad-hoc and error-prone process. It relies heavily on system administrators' ability to remember details of mechanism capabilities and interactions, and to correctly interpret high-level policies via mechanism configurations. Configuration mistakes are common, leading to surprises and, sometimes, significant vulnerabilities. As systems and networks expand and enforcement mechanisms are added, moved or upgraded, administrators are challenged to ensure that high-level policies are still being enforced consistently. Configuring enforcement mechanisms to protect larger, dynamic, and complex distributed systems is very difficult, as the configuration process does not scale well.

While manual security configuration is complex, labor intensive and error-prone for individual organizations, it is positively intractable for coalitions of multiple organizations with multiple systems and networks, cross-organizational resource sharing, a broad user base, and limited inter-organizational trust. Implementing security policies for collaborating organizations requires the configuration of fine-grained access control mechanisms to differentiate coalition participants from non-participants. For coalition operations, both the high-level security policy and the mechanisms for implementing that policy are more complex, increasing the likelihood of errors. Without tools to support the accurate configuration of enforcement mechanisms for complex and dynamic distributed systems, security concerns will continue to inhibit collaborative resource sharing.

The Configuration Synthesis and Policy Enforcement (SPiCE) project was funded by DARPA ATO under the Dynamic Coalitions program. The coalition focus of our sponsors highlighted the need for rigor and automation in our approach. To address the problem of error-prone, complex configurations, we developed a prototype system for automatically generating configurations for access policy enforcers and proved the correctness and completeness of our translation. We developed a language and tools to assist administrators in writing and refining coalition-based access control (*CBAC* [CTWS02]) policies, a compiler to translate *CBAC* policies into configurations for heterogeneous access policy enforcement mechanisms, and mechanisms to capture distributed system characteristics and deploy configurations to enforcers. To ensure the correctness of our compiler, we constructed a formal theory to define the semantics of our high-level policies and low-level mechanism configurations and prove that our translations preserve those semantics. Our most important results are very general, identifying characteristics of access control systems and policies that indicate the feasibility of using our translation approach. Thus, given a new access policy enforcement mechanism, we have tests for determining whether the SPiCE system can be extended to target that new mechanism. These general results extend the impact of our formal translation framework beyond *CBAC* and the suite of enforcement mechanisms targeted by our prototype.

In designing the SPiCE translation system, we bounded the scope of our investigation with the following assumptions. We begin with an assumption that the principal organizations participating in a coalition are autonomous: they can enter into agreements with other principal organizations; they have resources to protect; and they do not fully trust other principal organizations to protect those resources. Thus, a principal organization implements a policy to protect its own resources, using its own enforcers, which are configured and controlled by its own system administrators. We next assume that any negotiations among principal organizations have been done in advance and out of band. Thus, treaties, contracts or memoranda of agreement are in place and we can begin to apply technology to implement those existing agreements.

These two assumptions lead us to the following operational scenario: A policy maker for a principal organization takes a resource sharing agreement and codifies that agreement (via a SPiCE policy editor GUI) in our coalition policy specification language at the enterprise-level. Additional policy makers or administrators refine that high-level policy to lower levels of abstraction. When the policy is sufficiently refined, an administrator invokes the SPiCE compiler to translate the abstract policy into configurations for the principal organization’s access policy enforcers. The administrator may then deploy those configurations to the target enforcers, after which the new policy is in force. In this paper, we describe the formal semantics of translation, the safety and completeness results, the generalization of the results to additional enforcers, and the structure of the SPiCE system prototype.

The remainder of the paper is structured as follows. Section 2 summarizes background information on our coalition assumptions, the CBAC access control models, and the characteristics of our target access policy enforcement mechanisms. Section 3 defines the Cape access policy language for the CBAC_{basic} access control model. Cape is the high-level source language for our compiler. The architecture and implementation of the SPiCE system is discussed in Section 4. Section 5 presents our formal model of access control systems, along with proofs of our compiler’s correctness and completeness. Section 6 covers some related work in the areas of policy translation and composite access control systems. Section 7 provides a discussion of our results and future work. We end with a brief conclusion in Section 8.

2 Background

The SPiCE translator takes a high-level coalition-based access policy together with specifications of the distributed system structure and generates consistent configurations for three access policy enforcement targets. When we planned the SPiCE project, we selected our suite of target access policy enforcers with two primary criteria in mind: (1) Diversity. To be convinced that we could compile a high level access policy to a suite of heterogeneous policy enforcers, we needed to work with enforcers that were “different” from one another; (2) Ease of prototyping. The SPiCE compiler is a proof-of-concept prototype. We wanted to know as soon as possible if our approach to policy translation was infeasible. Thus, we avoided targeting enforcement mechanisms for which developing a configuration update mechanism would present significant engineering challenges (e.g., file systems, DBMS’s).

While our two criteria necessitated some trade-offs, we chose to target the Smart Firewalls system [BCG⁺01], developed by Telcordia Technologies together with two existing prototype implementations of object-oriented domain and type enforcement (OO-DTE). These prototypes, a web servlet and a modified Java runtime, were developed under the Secure Virtual Enclaves project [SYJS01]. There are significant differences between the OO-DTE enforcers and the Smart Firewalls system. The OO-DTE enforcers protect resources, while the Smart Firewalls system protects communication links between subjects and resources. The OO-DTE enforcers handle subjects representing users, while the Smart Firewalls system handles subjects and resources that are hosts in a network. The access control models are different, too. The OO-DTE mechanisms enforce policies in a simple role-based access control (*RBAC*) model [SCFY96], while the Smart Firewalls system enforces identity-based policies, where identities are host names (which map to IP addresses). The two OO-DTE enforcers are somewhat different from one another, as well. The Java RMI enforcer is able to mediate access to specific methods on Java objects that are accessed remotely, while the

web servlet enforcer mediates `http` requests to `get` specific files.

Updating policies is reasonably simple for each of the three enforcers. The OO-DTE enforcers were designed to accept policy updates over Java RMI. Our partners at Telcordia provided an RMI interface to the Smart Firewalls policy engine, to enable queries and policy updates. This enabled us to build a simple policy distributor to push policy updates from the compiler out to the various enforcement mechanisms.

The remainder of this section provides background on coalitions and the CBAC access control models, as well as on our three policy enforcement mechanisms. References to more thorough treatments of these topics are also included.

2.1 Coalitions and CBAC

In our review of coalitions and the CBAC family of access control models, we borrow text and figures from [CTWS02], which defines the CBAC models via abstractions of coalition concepts.

Organizations form coalitions to address common goals, sharing information system resources as necessary throughout their joint operations. These partner organizations are autonomous and have limited trust in one another. For example, commercial coalitions may implement supply chain arrangements, outsourcing or subcontracting relationships. Military alliances and joint task forces are formed to support joint defense objectives and peace-keeping efforts. Other coalitions form to support common humanitarian, environmental, or trade-related goals. In all of these diverse situations, organizations must control access to the resources they share with their coalition partners. Firewalls that distinguish only between outsiders and insiders are too coarse-grained to control inter-organizational sharing. While fine-grained access control is necessary, such control is currently complex to administer. CBAC was developed to reduce the complexity of administering fine-grained access control in coalition environments. CBAC models coalition access requirements, reducing administrative complexity through explicit representation of coalition concepts and organizational structure. The CBAC models capture resource sharing agreements among organizations and support delegation of authority through lines of organizational structure.

The CBAC family of access control models captures the entities involved in coalition resource sharing and identifies the interrelationships among those entities. The CBAC models range in expressivity and concomitant implementation complexity. The SPiCE project implemented the simplest of the models, $CBAC_{\text{basic}}$, which layers coalition access control concepts on top of a simple RBAC model. We briefly summarize here the aspects of $CBAC_{\text{basic}}$ that are necessary to understand the SPiCE work, using text and figures from [CTWS02]. More detail on the CBAC models can be found in [CTWS02].

The $CBAC_{\text{basic}}$ domain model is informally described in three views: the coalition, organization, and operations views. These views reflect the different perspectives of various coalition stake-holders on coalition activities and resources.

2.1.1 Coalition-level Domain Entities

Coalition entities are defined at the executive level where international diplomacy or top-level corporate agreements and their associated abstractions are specified.

A *coalition* is a collection of partner organizations joined together voluntarily and, often, temporarily, to accomplish one or more missions. A *partner organization* is an autonomous entity that is free to make agreements on its own behalf (e.g., a nation, a corporation). A *mission* is a goal that

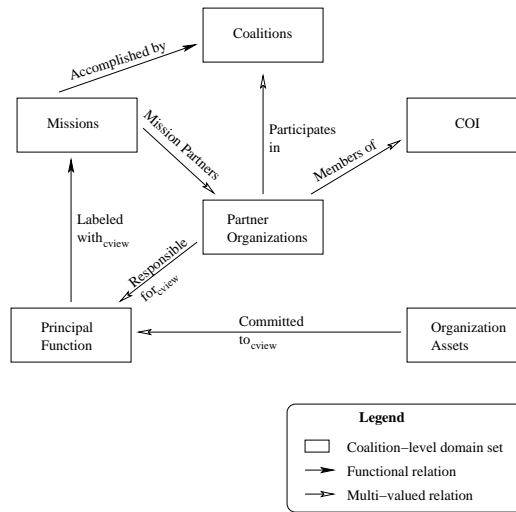


Figure 1: CBAC_{basic} Model, Coalition View

may be very broad or very specific. *Principal functions* are the fundamental classes of activities required to accomplish the mission. Partner organizations commit various *organization assets* to the accomplishment of those principal functions. Subsets of the larger coalition membership may form bi-lateral or multi-lateral *communities of interest (COI)*, based on common interests, or established levels of trust. Resource sharing among organizations within a COI may be more extensive than within the full coalition.

Figure 1 shows the coalition view of the domain model. Coalition entities appear as boxes in the figure, while relations between those entities are shown with arrows. Solid-headed arrows indicate functional relations, while open-headed arrows indicate multi-valued relations. [CTWS02] provides informal and formal definitions of the relations.

2.1.2 Organization-level Domain Entities

The organization view reflects the structures supporting coalition activities, as seen by management personnel within a partner organization. Organizational-level entities are defined at various levels within an organization (e.g., business unit, department, division). An organization entity might refine a coalition view entity or introduce a concept necessary to the implementation of coalition agreements.

The partner organizations, identified within the coalition view, may be subdivided into an arbitrary tree structure of *organizations*, permitting organizational hierarchies (e.g., a group within a department within a division) to be accurately modeled. Similarly, principal functions may be further segmented into a partially ordered structure of *functions*. Organization assets may be segmented into a partially ordered structure of *organization resources*. The ability to specify these hierarchical relationships supports the definition of a consistent hierarchy of views within an

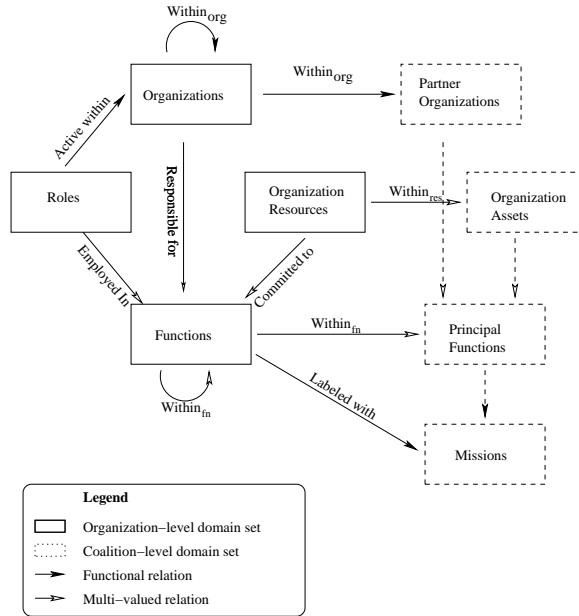


Figure 2: CBAC_{basic} Model, Organization View

organization: the group manager’s view shows resources that are within (i.e., components of) the resources that appear in the department manager’s view.

The role entity is introduced in the organization-level view. A *role* captures a coherent aspect of an individual’s job function within the organization. Figure 2 shows the organization-level view of the domain model. Organization-level entities appear as solid boxes in the figure, while related coalition-level entities appear as dashed boxes. In other words, portions of Figure 1 are shown again in Figure 2 to permit a convenient representation of relations that span the two views.

2.1.3 Operations-level Domain Entities

The operations view is concerned with the relationships between users and organization entities. Most security administration of computer systems is likely to take place at this level.

Only a single new coalition entity is introduced to the coalition domain model in the operations view: the user. The *user* is employed by an organization and performs in roles.

Figure 3 shows the operations-level view of the domain model. As before, portions of Figure 2 appear again in Figure 3 to permit a convenient representation of relations between “user” and organization-level entities. Note that we do not include an edge from users to organization resources in Figure 3, i.e., CBAC does not support a direct relationship between users and either functions or resources. This implies that CBAC access control models are fundamentally “role-based,” and an access control model built on the CBAC domain model will not support identity-based access policy statements. To add support for identity-based access control, one must modify the CBAC

domain model to specify an appropriately-constrained relationship between users and organization resources.

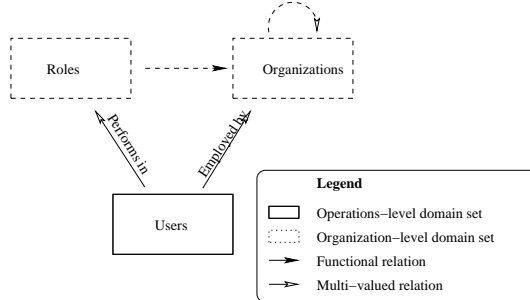


Figure 3: CBAC_{basic} Model, Operations View

2.2 Access Policy Enforcers

In the following sections, we focus on the access policy aspects of our enforcers, postponing discussion of system and implementation issues until Section 4.4, where we also describe the implementation of the SPiCE system.

2.2.1 OO-DTE Enforcers

In our review of OO-DTE policy structure, we borrow text from [SYJS01], which describes the use of the OO-DTE enforcers in the context of secure virtual enclaves.

Domain and Type Enforcement (DTE) [BSS+95] is a set of access control mechanisms for UNIX kernels, which extended Type Enforcement [BK85]. OO-DTE [STM+99] extends DTE to distributed object systems. DTE and OO-DTE define a mandatory policy in terms of the access rights of equivalence classes of subjects to equivalence classes of objects. An *object* is a resource accessed by software. A *subject* is a software component that accesses resources on behalf of a principal. Principals are persons or persistent programs (such as servers). In DTE, objects are grouped into equivalence classes called types, and subjects are grouped into equivalence classes called domains.

While the DTEL++ language [TSM98] for specifying OO-DTE access policies covers a wide range of issues in distributed object systems, such as class and type inheritance, polymorphism and abstraction, the prototype enforcers used for the SPiCE project implement the very simple Sveltd policy language. Sveltd policies have three components: type definition rules, which map distributed objects into types; domain derivation rules, which map subjects into domains; and an access matrix, operating on domains paired with types. Figure 4 shows an example Sveltd resource access policy.

A type definition identifies a collection of resources that will be treated identically for access control purposes. The example shows definitions for three types of resources: blueprints for two states, along with California water studies. For file objects accessed via a web server over https,

Type Mapping

```
florida_blueprints_t = https://army.mil/bridges/sunshine_state_skyway.gif,
                      https://army.mil/bridges/julia_tuttle_causeway.gif,
                      https://femaweb.gov/flblueprints/shared/*
california_blueprints_t = rmi://com.caweb.blueprinter view layer
california_water_studies_t = https://epa.gov/river_studies/*,
                             https://epa.gov/lake_studies/*,
                             https://epa.gov/ocean_studies/*
```

Domain Mapping

```
alice@army.mil = engineer_d
bob@army.mil = hydrologist_d
chris@fema.gov = engineer_d
```

Access Matrix

```
engineer_d = california_blueprints_t, florida_blueprints_t
hydrologist_d = california_water_studies_t
```

Figure 4: An OO-DTE Access Control Policy.

a type definition includes the URL of the file. In the example, Florida blueprint resources to be controlled include two specific .gif files, as well as the collection of files in the `flblueprints/shared` directory. For resources that are Java objects, a type definition includes the URL for the named Java object and, optionally, specific methods on that object. The `california_blueprints_t` type includes the `view` and `layer` methods on the Java `blueprinter` application for managing blueprints. The “_t” appended to each type name is a mnemonic device, rather than a syntactic requirement.

Domain derivation rules uniquely identify principals on whose behalf a subject will be assigned to a particular domain. Domains are, essentially, role specifications and enable a simple form of role-based access control (RBAC) [SCFY96]. Specifically, a subject is assigned to exactly one domain, there is no domain hierarchy, and there are no constraints on relationships between domains or the assignment of subjects to domains.

In the example, Alice and Bob (both members of the Army) are assigned to the *engineer* and *hydrologist* domains respectively. Chris (of FEMA) is assigned to the *engineer* domain. Alice and Chris, both *engineers*, will have identical access to resources, despite the fact that they belong to different organizations. Bob, the *hydrologist*, has access permissions that differ from Alice’s and Chris’. Thus, the semantics of “domain” within the context of DTE and OO-DTE are very different from the semantics of “domain” as defined in the networking literature.

Our implementation currently derives a domain based on the X.509 subject field value (an email address in our example). The “_d” appended to each domain name in the example is a mnemonic device, rather than a syntactic requirement. The access matrix shows the types of resources to

which subjects in a given domain are permitted access. In our example, *engineers* may access blueprints, while *hydrologists* may access water studies.

2.2.2 Smart Firewalls

The Smart Firewalls system manages configurations of network packet filters to support both positive and negative network connectivity policies. The system separates the specification of policy (definition of invariants) from the implementation of that policy (identification of necessary mechanism configuration). Information about network structure and devices, such as physical connectivity, IP address mapping, and routing tables, is provided to the Smart Firewalls system by a network management system (e.g., HP Openview or Columbia University’s NESTOR). The separation of invariants from implementation, together with the network structure information, enable dynamic reactions to changing network conditions (e.g., loss of connectivity or changes in network device configurations) via re-calculation of device configurations to maintain policy invariants.

The SPiCE system uses the Smart Firewalls Policy Engine to create packet filtering (iptables) rules. These rules allow the network connections enabling resource accesses permitted by high-level policies and disallow all other network connections. In Section 4.4, we discuss details of the SPiCE system interactions with Smart Firewalls.

Smart Firewalls policy is not specified in a textual language, but through XML specifications generated either by a GUI or by other programs such as the SPiCE compiler. For ease of reading, however, we write text-based example Smart Firewalls policy statements. For purpose of implementing SPiCE policies, we are interested in Smart Firewalls policy statements like the following:

ALLOW *Alice’s machine* (any port) to reach *Accounting subnet* (*http*)

DENY *Engineering subnet* (any port) to reach *Accounting subnet* (*all services*)

The Smart Firewalls policy engine can examine these new rules in the context of existing rules and the current network topology, and either generate firewall configurations to enforce the new rules or report errors. Depending on the network topology, there may be a number of routers and firewalls “standing between” *Alice’s machine* and the *Accounting subnet* that she needs to reach. Smart Firewalls configures all of the necessary devices on the multiple paths available to enable *Alice* to make web requests of hosts in the *Accounting* department. If there are no firewalls to mediate access between the *Accounting* and *Engineering* subnets, then Smart Firewalls cannot enforce the second rule and will report an error. Alternatively, if *Alice’s machine* is on the *Engineering subnet*, then we have a policy conflict, which would also be reported.

[BCG⁺01] contains more details about the Smart Firewalls system, while [KYBR99] discusses the NESTOR network management’s system use of Smart Firewalls.

3 The Cape Access Policy Language for CBAC_{basic}

CBAC_{basic} is a coalition-focused access control model that layers coalition access control concepts on top of a simple role-based access control (RBAC) model. CBAC_{basic} captures entities involved in

coalition resource sharing and identifies the interrelationships among those entities. In this section, we introduce the Cape language for expressing CBAC_{basic} access control policies.

Cape was designed to improve the scalability of access control administration. Though administrative scalability is also a significant issue within individual organizations, coalition operations compound the problem with multiple organizations, each with many users and many resources. Cape takes a two-pronged approach to supporting administrative scalability by providing: (1) coalition-focused domain and policy abstractions; and (2) a refinement process for consistently stepping down through levels of policy abstraction. By providing abstractions and supporting refinement, Cape allows administrators to write access control policies that are understandable within the coalition context and ensure that lower-level policy statements implement the top-level abstractions consistently.

The Cape policy expression language borrows much of its syntax from the Ponder policy expression language [DDLS01]. The Ponder language supports policies for authorization, delegation, information filtering, and refrain. Cape focuses more narrowly on strictly positive authorizations, but expands the content of an authorization to support coalition entities captured in CBAC_{basic}: COALITIONS, COI (Communities of Interest), MISSIONS, PARTNERORGANIZATIONS, PRINCIPALFUNCTIONS, ORGANIZATIONASSETS, ORGANIZATIONS, ROLES, FUNCTIONS, and ORGANIZATIONRESOURCES.

In the remainder of this section, we provide an informal introduction to Cape, define notions of policy refinement, consistent and completed policies, give an example based on U.S. Federal Emergency Management Agency (FEMA) coalitions, and offer a formal syntax specification of the language.

3.1 Components of Cape

In this section, we discuss Cape policy statements and authorizations (which act as atoms of policy), simple expressions for building policies from these atoms, refinement rules that relate the atoms of policy to one another, and definitions of consistent and completed.

3.1.1 Policy Statements

A Cape policy statement defines the scope of access that a coalition-level subject may be permitted to a target via a given action and with specific intent. Based on the CBAC_{basic} model, a *coalition-level subject* may be a PARTNERORGANIZATION, a COI (i.e., a set of PARTNERORGANIZATIONS), or an ORGANIZATION (a sub-organization of a PARTNERORGANIZATION). Since organizations do not directly access resources – only subjects that are software processes acting on behalf of USERS access resources – a Cape policy statement does not directly authorize any specific access. Rather it bounds the scope of access that sub-ORGANIZATIONS or USERS might be permitted. For example, a CEO may announce that company employees will be able to access paycheck stubs online. Without additional information and authorizations, this high level “policy statement” does not allow employees to take any specific actions. As we will later see, Cape policy statements must be refined into specific authorizations before software processes are permitted to access system resources. In our example, an employee must receive a user ID and default password from the human resources department before she can access her paycheck stub online.

A *target* is an ORGANIZATIONRESOURCE. (An ORGANIZATIONASSET is a special ORGANIZATIONRESOURCE that is not a component of any higher-level ORGANIZATIONRESOURCE). An *action* is a method

or PERMISSION by which the target may be accessed. For example, one might `read`, `write` or `execute` a target that is a file. A target that is a database relation might be `selected` or `joined` with another relation.

Finally, an *intent* is a FUNCTION. (A PRINCIPALFUNCTION is a special FUNCTION that is not part of any higher-level FUNCTION). The CBAC_{basic} domain model includes functions relating FUNCTIONS, MISSIONS and COALITIONS, so that by specifying a FUNCTION, we know the associated MISSION and COALITION. For example, an ORGANIZATION might be responsible for the `payroll` FUNCTION for the `general operations` MISSION of the `small business co-op` COALITION. The Cape policy statement of Example 3.1 for `payroll checks` allows the `accounting department` to generate check orders for purposes of payroll.

Example 3.1 (Payroll Checks Policy Statement)

```
Policy Statement payroll checks{
    Coalition Subject accounting department
    Target          check order
    Action          {generate}
    Intent          payroll
}
```

3.1.2 Authorizations

Cape policy statements bound the scope of allowable access but do not directly specify that an access is permissible. A Cape authorization denotes that a subject acting in a specific ROLE is permitted to access a target via a given action and with specific intent. In a Cape authorization, the *subject* must be a ROLE. We do not permit a USER to act directly as a subject. Rather, the CBAC_{basic} domain specification assigns USERS to ROLES, which may then appear in authorizations.

Targets, actions and intents are specified for authorizations just as they are for policy statements. In Example 3.2, a USER in the `accountant` ROLE is permitted to `generate check orders` for purposes of payroll.

Example 3.2 (Payroll Accountant Authorization)

```
Authorization payroll accountant{
    Subject accountant
    Target check order
    Action {generate}
    Intent payroll
}
```

3.1.3 Policies

Multiple policy statements and authorizations are usually necessary to describe fully the access policy intended by an ORGANIZATION for its collection of ORGANIZATIONASSETS and ORGANIZATIONRESOURCES. Cape provides two policy constructs to support the aggregation of multiple statements: the Declare Policy statement creates a named policy object, while the assignment statement assigns named authorizations to a policy object. For example, the accounting policy contains policy statements and authorizations that govern access to a variety of accounting

resources. In the statements below, the “+” and “-” operators work in the usual way to add or remove policy statements and authorizations to or from the policy definition.

Example 3.3 (Accounting Policy)

Declare Policy accounting;

```

accounting := accounting + payroll checks;
accounting := accounting + payroll accountant;

accounting := accounting + engineering specs;
accounting := accounting - engineering specs;

```

3.1.4 Refinement

To ensure that coalition policies implement top-level organizational intentions, we require that every authorization be traceable to a policy statement for a PARTNERORGANIZATION to use an ORGANIZATIONASSET. This prevents mid- (or low-) level policy makers from introducing policy statements “out of the blue,” without consideration for high-level organizational agreements. In this section, we define a refinement constraint to capture the traceability requirement. For every policy statement and authorization in a policy there must be a refinement path whose root is a top-level policy statement. The highest level policy statements do not refine any other statements, so they may be included in any policy. Note that because Cape policy statements are strictly positive, top-level policy statements cannot conflict with one another.

The conditions for a refinement relationship to hold between two policy statements are based on relationships between the pairs of subjects, targets, actions and intents. In the CBAC_{basic} model, these relationships are named **Within**_{org}, **Within**_{res}, **Within**_{perm}, **Within**_{fn}, and **ActiveWithin**, where the subscripts of the **Within** relations refer to ORGANIZATION, RESOURCE, PERMISSION, and FUNCTION. These relations are defined precisely by the CBAC_{basic} model in [CTWS02].

Informally, each of the **Within** relations corresponds to a notion of containment, imposing a partial order on the set of entities it relates: a sub-ORGANIZATION is **Within**_{org} its parent ORGANIZATION; a given RESOURCE may be a sub-component of a larger RESOURCE (a record is **Within**_{res} a relation which is also **Within**_{res} a database); a refining PERMISSION is **Within**_{perm} a refined PERMISSION when the refined PERMISSION is necessary and sufficient for the refining PERMISSION (e.g., read PERMISSION on file F can be refined by a select PERMISSION on the database relation contained within F); a sub-FUNCTION is **Within**_{fn} a super-FUNCTION with broader scope. A ROLE is **ActiveWithin** a specific ORGANIZATION, since ROLES are not expected to be uniformly defined by every ORGANIZATION within the COALITION, rather each is defined and maintained by a specific ORGANIZATION. Finally, an ORGANIZATIONASSET is a RESOURCE that is not **Within**_{res} any other RESOURCE.

We use the notion of containment in defining refinement for policy statements and authorizations. The target of the refining statement must reside **Within**_{res} the target of the refined statement. Similarly, the subject of the refining statement must reside **Within**_{org} the subject of the refined statement or the subject of the refining statement must be a ROLE that is **ActiveWithin** the ORGANIZATION of the refined statement. The PERMISSIONS specified for the refining statement must be appropriately **Within**_{perm} the PERMISSIONS of the refined statement. Finally, the

FUNCTION specified in the intent of the refining statement must be **Within_{fn}** the FUNCTION specified in the refined statement. We can now formally define refinement and related characteristics of policy statement, authorizations and policies.

Definition 3.4 (Refines) *A policy statement or authorization $A_2 = (s_2, t_2, P_2, i_2)$ refines policy statement $A_1 = (s_1, t_1, P_1, i_1)$ when all of the following four properties hold:*

1. *Subject containment. One of the following holds:*
 - s_2 is a ROLE and s_1 an ORGANIZATION such that **ActiveWithin**(s_2) = s_1 , or
 - s_1 and s_2 are ORGANIZATIONS such that **Within_{org}**(s_2, s_1).
2. *Target containment. t_1 and t_2 are ORGANIZATIONRESOURCES such that **Within_{res}**(t_2, t_1).*
3. *Action containment. $P_1 = \{p_1, p_2, \dots, p_n\}$ and $P_2 = \{p_{n+1}, p_{n+2}, \dots, p_{n+m}\}$ where the p_i are PERMISSIONS and **Within_{perm}**(P_2, P_1).*
4. *Intent containment. i_1 and i_2 are FUNCTIONS such that **Within_{fn}**(i_2, i_1).*

The notion of refinement can relate one statement to another, allowing us to define chains of statements and authorizations that capture delegation from high-level organizations, through smaller organizations, eventually down to specific roles:

Definition 3.5 *An authorization path is a sequence of policy statements and authorizations (a_1, a_2, \dots, a_n) where each a_i refines a_{i-1} .*

3.1.5 Consistent and Completed Policy

We can now use the definition of refinement to ensure a consistent and completed policy. Policies must be both (1) rooted in decisions made at the highest organizational levels; and (2) implemented through successive refinements to a sufficiently concrete level to support access mediation for user requests. To satisfy these two requirements, we define conditions on policy statements in a path:

Definition 3.6 (Abstract Policy Statement) *A policy statement $A = (s, t, P, i)$ is abstract when all of the following three properties hold:*

1. s is a PARTNERORGANIZATION, and
2. t is an ORGANIZATIONASSET, and
3. i is a PRINCIPALFUNCTION.

An abstract statement is written at the “highest level” possible within the specified domain. By requiring that a path begin with an abstract statement, we ensure upward traceability of low-level statements and authorizations: middle- or low-level organizations cannot introduce statements or authorizations “out of the blue.”

Definition 3.7 (Consistent Authorization Path) *An authorization path (a_1, a_2, \dots, a_n) is consistent when a_1 is abstract.*

Definition 3.8 (Concrete Authorization) *An authorization $A = (s, t, P, i)$ is concrete. Note that for an authorization, s is a ROLE.*

Because the $\text{CBAC}_{\text{basic}}$ model is inherently role-based, we require that an authorization specify a user ROLE in the subject, before resource access can be granted. Thus, for example, a policy-maker for the `small business co-op` may write an abstract authorization granting the organization `small business co-op` access to its `check orders`, but without refinement to a specific ROLE, no `co-op` personnel will be permitted to access that data. One could immediately write an authorization allowing anyone in the ROLE `co-op member` to access the data, but that might be an unreasonably broad authorization. We prefer, therefore, to create a sequence of policy statements for coalition subjects that are successively smaller sub-organizations and end the sequence with an authorization whose subject is a ROLE like `accountant`. The system can then permit any USER that **PerformsIn** the ROLE `accountant` to access the `check orders`. Placing our two requirements onto authorization paths, we have that:

Definition 3.9 (Completed Authorization Path) *An authorization path (a_1, a_2, \dots, a_n) is completed if a_1 is abstract and a_n is concrete.*

The definition of a completed Cape policy can then be stated as:

Definition 3.10 (Completed Policy) *A completed policy is a set of completed authorization paths.*

Of course, we expect that policies will be composed over time. High-level statements will be successively refined and interpreted at lower and lower levels by various organizations until concrete authorizations for user roles are generated. Therefore, we need a definition for a policy that is not wrong, just incomplete:

Definition 3.11 (Composing Policy) *A composing policy is a set of consistent authorization paths.*

3.2 Example

In this section, we illustrate a Cape policy specification based on coalition-based efforts of the U. S. Federal Emergency Management Agency (FEMA) [FEM01]. FEMA’s strategic plan [FEM00] is defined in terms of the coalitions, both standing and ad hoc, in which it participates.

In a wide variety of contexts, FEMA cooperates with and coordinates the activities of federal, state and local agencies together with the American Red Cross and other non-governmental organizations. Detailed, FEMA-based examples of a coalition structure using the $\text{CBAC}_{\text{basic}}$ model appear in [CTWS02]. Project Impact is a FEMA-led collection of community-focused coalitions for “disaster resistant communities.” Our example adds fictional details to a coalition effort to perform risk assessment and mitigation for the 1998 flooding on the Pajaro River in California.

Figure 5 shows a portion of a Cape policy that might have been created by FEMA to describe the access policy for the Pajaro River risk mitigation resources. The policy shows two abstract policy statements at the top of the figure: `Top Army`, and `Top FEMA`. The U.S. Army has signed a memorandum of agreement with FEMA to participate in Project Impact and provide engineering assistance. FEMA policy makers write the `Top Army` policy statement to assert that some

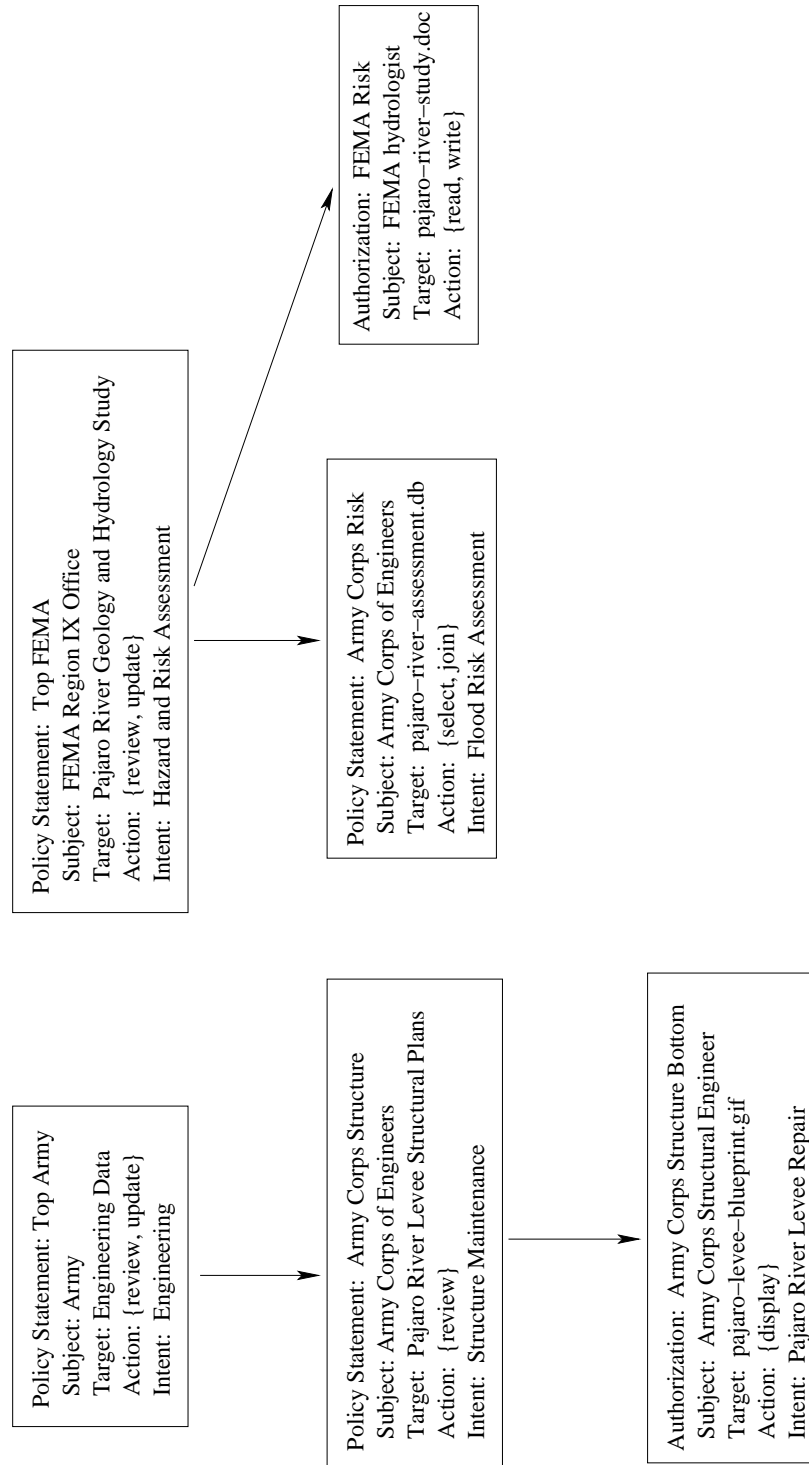


Figure 5: FEMA Project Impact: Pajaro River Risk Mitigation Policy.

Army personnel should have access to **Engineering Data** to perform an **Engineering** function (for the **Pajaro River Flood Mitigation** mission of the **Project Impact** coalition). That statement is refined into the **Army Corps Structure** policy statement below. In this statement, the sub-organization **Army Corps of Engineers** is offered some scope for accessing the **Pajaro River Levee Structural Plans**. Finally, a concrete authorization, **Army Corps Structure Bottom**, authorizes users in the role of **Structural Engineer** for the **Army Corps** to **display** the **pajaro-levee-blueprint.gif** file. This sequence of three statements forms a consistent and completed authorization path, as does the path from the **Top FEMA** policy statement to the **FEMA Risk** authorization. Note that the path from **Top Army** to **Army Corps Risk** does not terminate in a concrete authorization. Because this path is not completed, the **Pajaro River Risk Mitigation** policy is composing, rather than completed. When a concrete authorization is added below the **Army Corps Risk** statement, the policy will be completed and deployable.

3.3 Syntax

In this section, we provide the BNF definition of Cape syntax.

The set of terminal symbols is:

- **Declare Policy, Authorization, Policy Statement, Coalition Subject, Subject, Target, Action, Intent**,
- symbols embraced within “ ” (double quotes), and
- letters and digits

The following BNF meta-symbols are used in the productions:

- ::= means “is defined as”
- | means “or”
- { and } mean “repeated zero or more times”

The following BNF productions define Cape syntax. The Cape start symbol is `cape_policy`.

```

cape_policy      ::= p_declare
                  p_implement { p_implement }
p_declare       ::= Declare Policy d_name “;”
d_name          ::= name
p_implement     ::= d_name “:=” d_name “+” p_authorization “;”
                  | d_name “:=” d_name “-” p_authorization “;”
                  | d_name “:=” d_name “+” p_statement “;”
                  | d_name “:=” d_name “-” p_statement “;”
p_authorization ::= Authorization s_name “{”
                  coalition_subject
                  target_expression
                  action_expression
                  intent_expression

```

```

    “}” “;”
p_statement ::= Policy Statement s_name “{”
             role_subject
             target_expression
             action_expression
             intent_expression
             “}” “;”

s_name      ::= name
coalition_subject ::= Coalition Subject c_sub {c_sub}
c_sub       ::= partner-org | CoI | org
role_subject ::= Subject role {role}
target_expression ::= Target t_element {t_element}
t_element     ::= t_resource | t_asset
t_resource    ::= org-resource
t_asset       ::= org-asset
action_expression ::= Action “{” a_action {a_action} “}”
a_action      ::= operation
intent_expression ::= Intent i_constraint
i_constraint   ::= function

function      ::= identifier
name          ::= identifier
partner-org   ::= identifier
CoI           ::= identifier
org           ::= identifier
role         ::= identifier
org-resource  ::= identifier
org-asset     ::= identifier
operation     ::= identifier
identifier    ::= letter { letter | digit }

```

4 The SPiCE System

The SPiCE system is fundamentally a multi-target compiler with auxiliary components to acquire the source to be translated and then distribute the resulting synthesized configurations to the target enforcement mechanisms. We first give a conceptual overview of the system components, describe the Virtual System Model and translation algorithms, then discuss the system architecture and implementation.

4.1 Overview

Figure 6 shows the conceptual components of the SPiCE translation system. The first input to the system is the CBAC domain specification. As discussed in Section 2.1, a specification of a CBAC

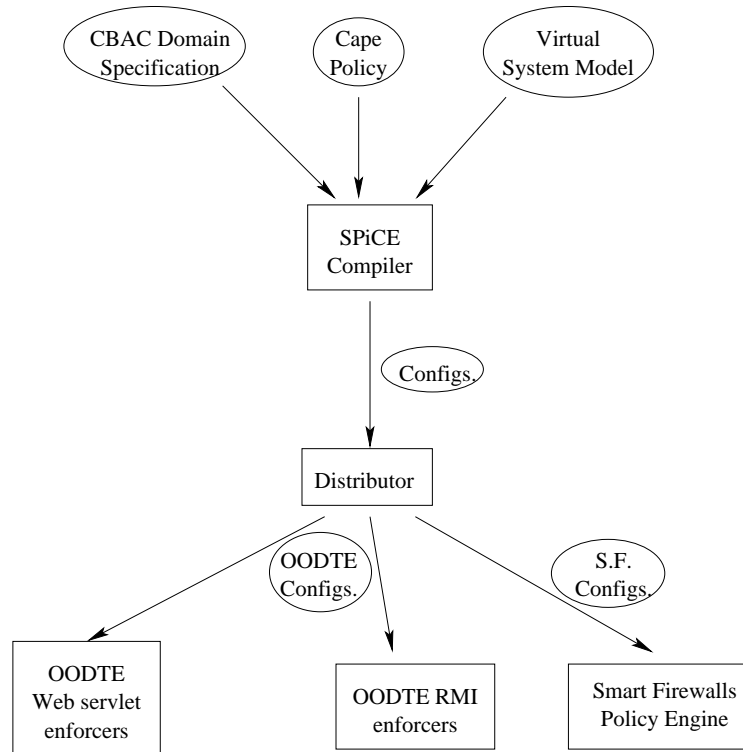


Figure 6: Conceptual SPiCE Policy Translation

model for a given coalition environment includes:

- Sets of COALITIONS, FUNCTIONS, ORGANIZATIONS, ROLES, ORGANIZATIONRESOURCES, etc.,
- Specification of the relations that hold between those sets. From the example of Section 3.2 the **ROLE Structural Engineer** is **ActiveWithin** the **ORGANIZATION Army Corps of Engineers**, and
- Constraints on the relations between CBAC entities.

The compiler uses the CBAC ORGANIZATIONS, ROLES, and FUNCTIONS to generate OO-DTE Domains (see Section 2.2.1 for details) and the CBAC ORGANIZATIONALRESOURCES and PERMISSIONS to generate OO-DTE Types.

Of course, the Cape policy is also an input to the compiler. As discussed in Section 3, a concrete Cape authorization includes a ROLE, ORGANIZATIONRESOURCE, PERMISSIONS, and FUNCTION. The SPiCE compiler will use the concrete Cape authorizations, together with the OO-DTE Domains and Types previously generated, to build the OO-DTE access control matrix.

The final input to the SPiCE system is the Virtual System Model, which provides an abstraction of the distributed system in which processes make requests on behalf of users for system resources and access policy enforcers mediate those access requests. The Virtual System Model effectively identifies which enforcers mediate access requests for which pairs of subjects and resources. Section 4.2 describes this model in more detail.

After the SPiCE compiler has synthesized configurations for OO-DTE and for Smart Firewalls, it forwards them to the Distributor for distribution to the enforcers. Each enforcer is capable of accepting configurations via a programmatic interface in addition to the GUIs it provides for stand-alone use. On receiving the updated configurations, the OO-DTE enforcers will begin using them to calculate responses to access requests. The Smart Firewalls system, however, will perform its own compilation process on the configurations provided and will generate and distribute iptables rules for the Linux packet filters it manages.

4.2 Virtual System Model

The Virtual System Model captures those characteristics of the underlying distributed system that the SPiCE compiler requires for synthesizing configurations. Fundamentally, the SPiCE compiler needs only to know which requests can be mediated by which enforcers. Figures 7, 8 and 9 show a portion of the Virtual System Model. In Figure 7, we show a system subject and a system resource, where system subjects make requests for system resources. In our model, a system subject can be “viewed” through a number of facets (our prototype implements two, the X.509 and Network facets), depending on what type of software is doing the “viewing.” For example, when handled by networking software, a system subject is a host with ports through which it might send messages using various protocols over the network. When handled by middleware software, however, that same system subject might be viewed as a principal, that is represented by data carried in an X.509 identity certificate (e.g., subject name, public key, organization). System resources are similarly “viewed” differently by software in the distributed system. For example, a Java application would identify a system resource by naming a Java object and a method to execute on that object. This is modeled by the middleware facet of the system resource.

By modeling system subjects and resources as multi-faceted entities, we capture essential relationships across multiple system layers, and among any number of applications that must interact throughout the system. These abstractions significantly reduce the complexity of mapping abstract CBAC entities to system entities in the translation process.

Our Virtual System Model for requests as shown in Figure 8 extends the models for system subjects and resources. Each request specifies a system subject requester and a system resource that is requested. For example, when a user asks the distributed system to provide some data, networking software views this as a request to pass messages from the sending host to the host on which the resource resides (and perhaps through several hosts along the way). A web server may interpret that same request as that of a principal with credentials contained in an X.509 certificate attempting to access data stored in a file, maintained by the file system of the host on which the web server runs. Thus, our model links the abstract notion of a “distributed system request” to a number of concrete requests handled by the various pieces of system and application software.

Finally, Figure 9 shows our Virtual System Model for our three access policy enforcers. Our OO-DTE-based web servlet enforcers take web requests (distributed system requests, viewed through a web facet) and mediate them according to the OO-DTE access control model. Of course, one

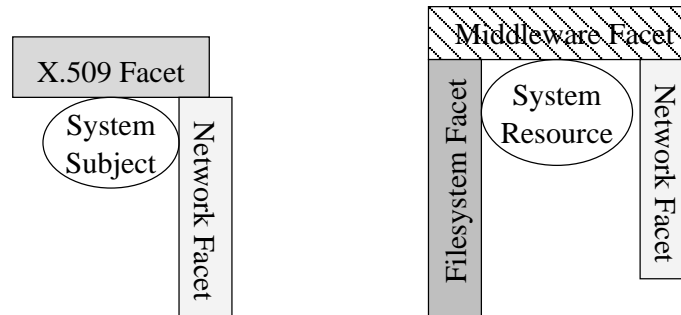


Figure 7: Virtual System Subject and Resource

could build a different web servlet enforcer that took those same web requests, but mediated access according to an identity-based access control model (IBAC). A bank might employ just such an approach to mediate access to specific user accounts. Our OO-DTE RMI enforcers also mediate access to methods on Java objects, using the OO-DTE access control model. Finally, the Smart Firewalls system, because of the packet filters that implement its rules, enforces access based on a network IBAC model, where the identity in question is that of the sending host.

4.3 Translation Algorithms

SPiCE translation is performed in three stages.

1. Compute equivalence relations on low-level entities: X.509 subjects, URLs, IP addresses, and ports. The equivalence relations are chosen so that any low-level entities that can be distinguished by Cape policy fall into different equivalence classes. These classes are used to build the *policy-independent* portion of the configuration, namely the OO-DTE domain and type maps and the Smart Firewalls logical network and service definitions.
2. Given the input Cape policy, compute the *policy-dependent* portion of the configuration, namely the OO-DTE access matrices and the Smart Firewalls policy. This is relatively straightforward, and involves checking if representatives of the equivalence classes computed in step 1 are enabled by the Cape policy.
3. Check if the configuration has enabled any additional CBAC requests not allowed by the Cape policy. If so, report an error. If not, the configuration is correct.

We now add some detail to the description of the three stages of translation:

1. Policy-independent configuration.
 - (a) Compute equivalence relations.

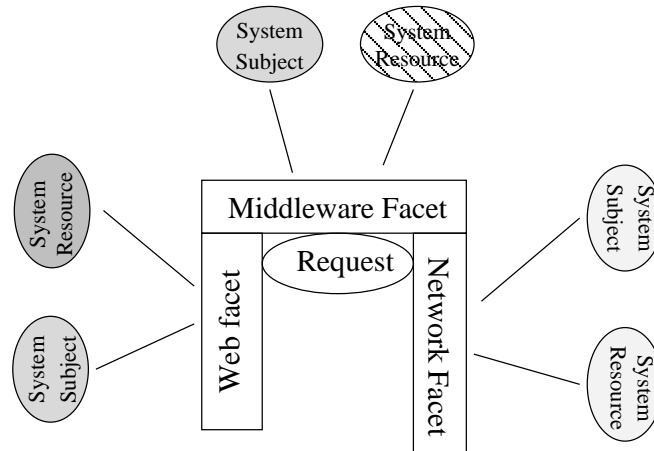


Figure 8: Virtual System Requests

- i. Compute equivalence relations on X.509 subjects, IP addresses, and ports based on whether CBAC ROLES can distinguish them.
 - ii. Compute equivalence relations on URLs, IP address, and ports based on whether CBAC RESOURCES can distinguish them.
- (b) Configure OO-DTE domain map and type map.
- i. Define a domain for each equivalence class of X.509 subjects. Define the domain map to map an X.509 subject to the domain corresponding to its class.
 - ii. Define a type for each equivalence class of URLs. Define the type map to map a URL to the type corresponding to its class.
- (c) Configure Smart Firewalls.
- i. Define a network name for each equivalence class of IP addresses based on CBAC ROLES and for each equivalence class of IP addresses based on CBAC RESOURCES.
 - ii. Define a service name for each pair of an equivalence class of ports based on CBAC ROLES and an equivalence class of ports based on CBAC RESOURCES.
2. Policy-dependent configuration.
- (a) Configure OO-DTE access matrix. An entry in the access matrix corresponds to an equivalence class of X.509 subjects and an equivalence class of URLs. Turn on an entry if some representative subject and URL is enabled by the Cape policy.
 - (b) Configure Smart Firewalls policy rules. A policy rule consists of a source network name, a target network name, and a service name. For each CBAC request, add policy rules

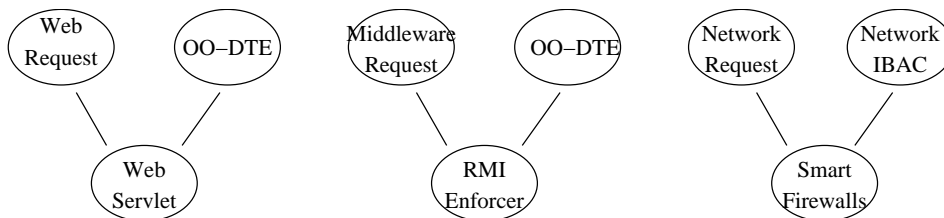


Figure 9: Virtual System Enforcers

with the source network name whose class corresponds to the CBAC ROLE, the target network name corresponds to the CBAC RESOURCE, and the service name corresponds to the pair of classes on ports, with the first component corresponding to the role and the second to the resource.

3. Correctness check. Check each CBAC request that is *not* in the policy to make sure that no corresponding low-level request is accepted.

4.4 Architecture and Implementation

To implement the conceptual translation process of Figure 6, we developed a Java-based prototype. Figure 10 depicts data and control flow in the system: the arrows show data flow, while control flows from the SPiCE controller (on the right) to the other shaded boxes in the diagram. An administrator would control the system using the GUI attached to the SPiCE controller. Each of the boxes in the diagram represents a Java class, while each of the ovals and half-ovals represent data that is manipulated by the translation system. The cylinder near the top of the diagram represents a relational DBMS (Postgres) that stores the CBAC domain specification, Cape policies, and a portion of the virtual system specification. Communication between distributed Java-based system components is accomplished over RMI, while any communication between these components and the DBMS is over JDBC.

We now discuss how the system of Figure 10 implements the conceptual translation process of Figure 6. The CBAC domain specification and Cape policies are stored in a database. Triggers encode the constraints of the CBAC model, as well as the refinement conditions for Cape policies. The Policy Editor is a tool to assist a policy maker in interacting with the DBMS to build a policy. This tool orders database queries and updates, constraining the policy construction and refinement processes and presenting only viable options to the administrator through a GUI to avoid errors that would cause the database triggers to fire. Though we did not construct a similar tool for building the CBAC domain specification or virtual system specification, such a tool will be necessary for realistic use of the translation system.

The Virtual System Model required by the compiler is constructed by the Network Oracle component of the SPiCE system. In the prototype, a portion of the data necessary to build that model

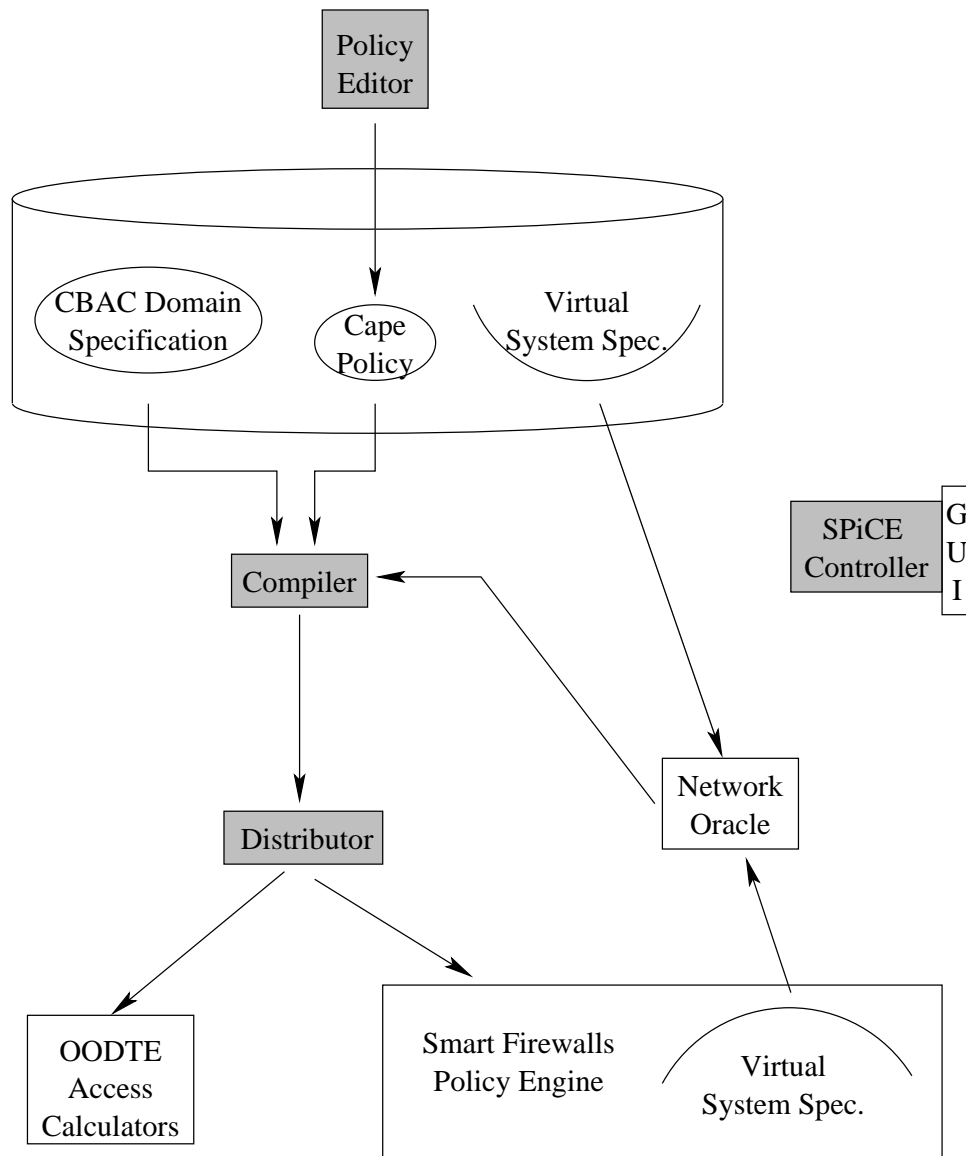


Figure 10: SPiCE System Data/Control Flow

is stored in the database, while additional material is acquired from the Smart Firewalls Policy Engine. The database portion includes information about the OO-DTE enforcers—specifically, the addresses of the hosts on which they run and the file paths of resources served by (and, thus, protected by) the web servlet enforcer. Information about network topology—specifically, subnets separated by packet-filtering routers or firewalls—is extracted from the Smart Firewalls Policy Engine. If our prototype were developed into a product, the Network Oracle would serve as an interface to a network management system (like HP Openview or CA Unicenter) from which it would construct the Virtual System Model. Finally, the SPiCE compiler translates the given Cape policy into Sveltdt policy, embedded in a Java object for OO-DTE, and Smart Firewalls policy, represented in serialized XML. Both are transmitted to their respective enforcers over RMI.

5 Formal Semantics of Translation

There are two properties that a policy compiler such as SPiCE should have. First, if the compiler succeeds in producing a configuration, we would like to know that that configuration accurately implements the input policy. Second, if the compiler fails to produce a configuration, we would like it to do so only because there is no configuration that implements the input policy. The first property is one of correctness, while the second is one of completeness.

Since policy translation typically deals with situations where security is essential, we would like to have as much certainty as possible that our compiler is correct and complete. Toward this goal, in this section, we present a formal model of access control systems and of policy translation between access control systems. Our model of access control systems and policies is quite general, and can express both high-level policy like Cape and low-level configuration of enforcers like OO-DTE and Smart Firewalls.

We model policy translation as a function mapping from high-level policy to low-level policy. We describe the relationship between the levels via a function mapping from low-level policy to high-level policy. The key to our approach is to prove that this pair of functions forms a *Galois connection*, which we will formally define later. Once we have proved our translation is a Galois connection, we can apply the *Galois representability lemma* (Lemma 5.17) to conclude that our translation is both correct and complete.

The use of a Galois connection reflects the intuition that low-level policy approximates high-level policy and that policy translation respects this relationship. The relationship is an approximation rather than that of inverses because, in some situations, low-level policy may not be able to make the distinctions required to exactly implement high-level policy. The loss of information happens because low-level enforcers operate independently and only have access to fragments of high-level requests, and hence can only distinguish requests that differ on the part they can see.

In Sections 5.1, 5.2, and 5.3, we define our model, define Galois connection, give examples of policy translation, and state the representability lemma. In Section 5.5 we apply the model to SPiCE, using the formal semantics of Cape, OO-DTE, and Smart Firewalls, expressed as an access control systems in our model. We then use general techniques for constructing Galois connections to build a Galois connection for SPiCE policy translation.

5.1 A Model of Access Control Systems

To build our model, we observe that all access control systems have the following in common: they take as input some form of request and produce as output a response, typically a yes or no, indicating whether the request was accepted or denied. So, we take a set of requests R as a basic parameter when modeling an access control system.

For example, to express Smart Firewalls in our model, we might define requests by

$$R = IP \times Port \times IP \times Port \times Protocol$$

Abstractly, a request to Smart Firewalls of the form $(ip_1, port_1, ip_2, port_2, prot)$ denotes a packet sent from source IP address ip_1 and source port $port_1$ to destination IP address ip_2 at port $port_2$ using protocol $prot$ (e.g. TCP). From our perspective, outside of Smart Firewalls, either the packet is allowed to go through or it is not. We don't care and don't need to know *how* Smart Firewalls configures the packet filters on the individual firewalls under its control. We simply care that once it has done so, a request (packet) will either be accepted or rejected.

As another example, for Cape, we might define requests by

$$R = Role \times Resource \times Methods \times Function$$

Here, a request is of the form $(role, res, ms, f)$, indicating that a user acting in role $role$ would like to apply methods ms to resource res for the function f . When we write Cape policy we don't care *how* the access control system decides whether such a request is accepted or rejected. We simply care that the yes/no answer given by the system corresponds to what our policy says. Of course, in implementing SPiCE we will care very much about the details of how the enforcers work together to correctly process the request.

By taking this abstract view, any access control system can be viewed equally well as the source or target system for policy translation. Further, it could have a direct implementation (as in OO-DTE) or an indirect one (as in Cape or Smart Firewalls)

As we have hinted in the above examples, a policy for some access control system determines which requests are accepted and which requests are rejected. From the point of view of an administrator writing policy or the point of view of a user attempting to access a resource, this set is often sufficient to understand the system's behavior. So, we define a policy to be a set of accepted requests.

Definition 5.1 (policy) *A policy on a set of requests R is a subset of R .*

We will use p to range over policy. For example, if $R = \{r_1, r_2, r_3\}$, then $p = \{r_1, r_2\}$ is a policy on R , and indicates that r_1 and r_2 should be accepted, while r_3 should be rejected.

An access control system always comes with some language for writing access control policies. For example, in Cape, that language consists of policy statements and authorizations. Once we have a policy language, we then write (formally or informally) rules that describe which requests should be accepted given a policy written in the language. These rules, or semantics, establish a very tight connection between this notion of policy and our formal definition of policy as a set of requests. This leads to the following definition.

Definition 5.2 (policy language) *A policy language for R is a pair $(L, d : L \rightarrow \mathcal{P}(R))$*

Intuitively, for $l \in L$, $d(l)$ gives the set of requests authorized by l . A policy language determines the policies that can be expressed. To formalize the idea of a set of policies that we can or would like to express, we define the notion of a *policy space*.

Definition 5.3 (policy space) *A policy space on R is a set of policies on R .*

For example, $\{\{\}, \{r_1\}, \{r_2, r_3\}\}$ is a policy space on $\{r_1, r_2, r_3\}$. In general, the policy space expressible by a policy language (L, d) is $\{d(l) \mid l \in L\}$.

When translating access control policy from a high-level system to low-level enforcers, the low-level enforcers induce a policy space on the high-level system (we will show precisely how later). In order for the translation to succeed, we would like for the induced policy space to be sufficiently expressive. This leads us to the following notion.

Definition 5.4 (implements) *Policy space ps' implements policy space ps if $ps \subseteq ps'$.*

Since policies are sets of requests, we can also take their set theoretic intersection and union.

$$\begin{aligned} \bigcap \{p_j \mid j \in J\} &= \{r \mid \forall j \in J. r \in p_j\} \\ \bigcup \{p_j \mid j \in J\} &= \{r \mid \exists j \in J. r \in p_j\} \end{aligned}$$

One property that is common to many policy languages is that if two policies p_1 and p_2 are expressible, then so is their union $p_1 \cup p_2$. We generalize this to policy spaces as follows.

Definition 5.5 (additive) *Policy space ps is additive if for all $ps' \subseteq ps$, $\cup ps' \in ps$.*

We say that policy language (L, d) is additive if the policy space expressible by (L, d) is additive. Many real-world policy languages are additive. In particular, in many languages, sometimes called positive access languages, a policy consists of a set of assertions, where each assertion determines a set (possibly a singleton) of requests to be accepted. In these languages, the set of requests accepted by the policy is the union of the sets of requests accepted by the individual assertions. Trivially, such a language is additive. For example, Cape is additive since we can take all the policy statements and authorizations from a collection of policies and combine them to form a single large policy. This combined policy will authorize exactly those requests accepted by the individual policies. As another example, Smart Firewalls is additive, since we can combine the Smart Firewalls policy rules from individual policies to form a single large policy. As an example of a policy space that is not additive, consider $\{\{\}, \{r_1\}, \{r_2\}, \{r_1, r_2, r_3\}\}$. This policy space is not additive because it does not include $\{r_1, r_2\}$.

A policy p *distinguishes* between two requests r_1 and r_2 either if it accepts r_1 and rejects r_2 , or vice versa. Given a policy space, we can consider whether a pair of requests can ever be distinguished by any policy in the space.

Definition 5.6 (indistinguishable) *r_1 is indistinguishable from r_2 in policy space ps if for all p in ps , $r_1 \in p$ if and only if $r_2 \in p$.*

Obviously, indistinguishability is an equivalence relation on requests. If r_1 is indistinguishable from r_2 in ps , we write $r_1 \equiv_{ps} r_2$. We write $[r]_{ps}$ for the equivalence class of r . That is, $[r]_{ps} = \{r' \mid r \equiv_{ps} r'\}$. We drop the subscript and write $r_1 \equiv r_2$ or $[r]$ if the context makes the policy space clear.

For any policy p in ps , an equivalence class of \equiv_{ps} is either wholly contained in p or is disjoint from p . This leads to the following definition.

Definition 5.7 (acceptable) *An equivalence class c of \equiv_{ps} is acceptable if there is some p in ps such that $c \subseteq p$. An equivalence class is unacceptable if it is not acceptable.*

If r_1 and r_2 are not accepted by any policy in a policy space ps then, trivially, $r_1 \equiv_{ps} r_2$. Hence, for any policy space ps , there is at most one unacceptable equivalence class of \equiv_{ps} . The following lemma tells us that every policy is determined by the acceptable equivalence classes that it contains.

Lemma 5.8 (unique decomposition) *For every policy space ps , every policy $p \in ps$ is uniquely expressible as a union of acceptable equivalence classes of \equiv_{ps} .*

On the other hand, we can consider a policy space in which every combination of acceptable classes determines a policy.

Definition 5.9 (complete) *A policy space ps is complete if it is additive and every acceptable equivalence class of \equiv_{ps} is in ps .*

For example, Cape is complete. The reason is that individual authorizations can be independently granted by an appropriate refinement. Hence, all requests are distinguishable and expressible.

When implementing a policy space ps , we must, at a minimum, be able to distinguish requests that ps distinguishes, as the following lemma shows.

Lemma 5.10 (implements respects equivalence) *If ps' implements ps , then for all r_1 and r_2 , if $r_1 \equiv_{ps'} r_2$, then $r_1 \equiv_{ps} r_2$.*

The converse of this lemma does not hold – that is, it is not sufficient to distinguish every pair of requests in order to implement a policy space. Consider $R = \{r_1, r_2, r_3\}$, $ps = \{\{r_1, r_2\}\}$, and $ps' = \{\{\}, \{r_2, r_3\}, \{r_3\}\}$. Notice that $\equiv_{ps'}$ is the identity relation, hence trivially we have that whenever $r_1 \equiv_{ps'} r_2$ that $r_1 \equiv_{ps} r_2$. However, ps' clearly does not implement ps . Also notice that ps' is additive, so even that is not enough to guarantee that we can implement ps .

Fortunately, for complete policy spaces, there is a simple sufficient condition that tells us when they can be implemented, at least by additive policy spaces.

Theorem 5.11 (additive implements complete) *If ps' is additive and ps is complete, then ps' implements ps if and only if ps' implements C , where C is the set of acceptable equivalence classes of \equiv_{ps} .*

This theorem tells us that to implement a complete policy space, we can concentrate on implementing the acceptable equivalence classes.

5.2 Direct Policy Translation

We now consider a simple example of policy translation in which requests of one access control system have a direct analogue in another. We model the source access control system via its set of requests R and the target access control system via its set of requests R' . We assume that we have a function $f : R \rightarrow R'$ that maps source requests to target requests. To explain the meaning of f , suppose we have some policy p' for the target system. What policy in the source system does p' represent? If p' accepts some request r' and there is some r such that $f(r) = r'$ then it seems clear that p' has accepted r . So, we can define the source policy corresponding to p' as $\{r \mid f(r) \in p'\}$. More generally, we can define a function up that maps policies on R' to policies on R . The reason

for the choice of the name up will become clear later.

$$\begin{aligned} up &: \mathcal{P}(R') \rightarrow \mathcal{P}(R) \\ up(p') &= \{r \mid f(r) \in p'\} \end{aligned}$$

Suppose that we would like to translate a policy p on R into a policy p' on R' , and have p' correctly implement p . We model our translator as a function $down$ (again, the naming will become clear later):

$$down : \mathcal{P}(R) \rightarrow \mathcal{P}(R')$$

What properties would we like $down(p)$ to satisfy? First, to ensure that $down(p)$ accepts enough requests, we would like that if policy p accepts some request r , then the image of r under f is accepted by the translated policy $down(p)$. Formally, if $r \in p$, then $f(r) \in down(p)$. We can ensure this by defining $down$ as follows.

$$down(p) = \{f(r) \mid r \in p\}$$

On the other hand we would like to ensure that $down(p)$ does not accept too many requests. So, if $down(p)$ accepts some request r' , then we would like *every* request r that maps to r' under f to be in p . That is, if $f(r) \in p'$, then $r \in p$. Combining the two requirements, we have:

$$r \in p \iff f(r) \in p'$$

Or, equivalently:

$$p = up(down(p))$$

Intuitively, $down$ describes how to *approximate* a policy on R as a policy on R' , while up describes the meaning of an approximate policy on R' as a full policy on R . Such a pair of functions, satisfying certain requirements, is called a *Galois connection*, which is a standard mathematical means used to formalize the notion of approximation.

Definition 5.12 (Galois connection) *Let U and D be partially ordered sets, $down : U \rightarrow D$, and $up : D \rightarrow U$. Then $(down, up)$ is a Galois connection between U and D if $down$ and up are monotone and if for all $u \in U$ and $d \in D$, $down(u) \leq d$ if and only if $u \leq up(d)$.*

Intuitively D is an approximation of U , $down(u)$ tells what details about u are irrelevant for the purposes of the approximation, and $up(d)$ gives the concrete meaning of d . The name $down$ is used to indicate the loss of information due to approximation, while the name up is used to indicate the increase in precision when expressing an approximate value in a more concrete way. We refer to the relationship between $down$ and up as the *reversal* property.

To apply Galois connections to direct policy translation, we take U to be the set of policies over R and D to be the set of policies over R' .

$$\begin{aligned} U &= \mathcal{P}(R) \\ D &= \mathcal{P}(R') \end{aligned}$$

As our partial order on policies, we use the natural partial order based on accepting more requests, that is, the subset order.

Definition 5.13 (policy order) $p_1 \leq p_2$ if $p_1 \subseteq p_2$.

This order leads to standard notions of safety and liveness. A policy is *safe* if it doesn't accept too many requests.

Definition 5.14 (safe) p_1 is safe with respect to p if $p_1 \subseteq p$.

A policy is *live* if it accepts enough requests.

Definition 5.15 (live) p_1 is live with respect to p if $p \subseteq p_1$.

We now show that we did indeed define a Galois connection.

Theorem 5.16 *The functions down and up, defined by*

$$\begin{aligned} \text{down} &: \mathcal{P}(R) \rightarrow \mathcal{P}(R') \\ \text{down}(p) &= \{f(r) \mid r \in p\} \\ \text{up} &: \mathcal{P}(R') \rightarrow \mathcal{P}(R) \\ \text{up}(p') &= \{r \mid f(r) \in p'\} \end{aligned}$$

are a Galois connection between $\mathcal{P}(R)$ and $\mathcal{P}(R')$.

Proof. We first show that *down* is monotone. Suppose that $p_1 \subseteq p_2$. Then,

$$\text{down}(p_1) = \{f(r) \mid r \in p_1\} \subseteq \{f(r) \mid r \in p_2\} = \text{down}(p_2)$$

We next show that *up* is monotone. Suppose that $p'_1 \subseteq p'_2$. Then,

$$\text{up}(p'_1) = \{r \mid f(r) \in p'_1\} \subseteq \{r \mid f(r) \in p'_2\} = \text{up}(p'_2)$$

Finally, we show the reversal property.

$$\begin{aligned} \text{down}(p) &\leq p' \\ &= \text{down}(p) \subseteq p' \\ &= \{f(r) \mid r \in p\} \subseteq p' \\ &= \forall r \in p. f(r) \in p' \\ &= \forall r \in p. r \in \{r_1 \mid f(r_1) \in p'\} \\ &= \forall r \in p. r \in \text{up}(p') \\ &= p \subseteq \text{up}(p') \\ &= p \leq \text{up}(p') \end{aligned}$$

□

The above construction is one of many standard ways of constructing a Galois connection; in particular it is a special case of Theorem A.13. In Appendix A, we describe many other ways to construct Galois connections, a number of which we use below.

In general, our approach is to model policy translation as a Galois connection between high-level policy $\mathcal{P}(R)$ and some approximation of it, D .

$$\begin{aligned} \text{down} &: \mathcal{P}(R) \rightarrow D \\ \text{up} &: D \rightarrow \mathcal{P}(R) \end{aligned}$$

In this section, we have shown a simple case, $D = \mathcal{P}(R')$. In subsequent sections, we will model other translations using more complex Galois connections. In all cases, we will define up to express the meaning of the target system as a source policy. The goal of policy translation will be to take a source policy p and find a $d \in D$ that implements p , that is, such that $\text{up}(d) = p$. We will do this by defining a down such that down and up form a Galois connection. We will then apply the representability lemma.

Lemma 5.17 (representability) *If (down, up) is a Galois connection, then $u = \text{up}(\text{down}(u))$ if and only if there exists a d such that $u = \text{up}(d)$.*

The consequence of this lemma is that we only need to consider $p' = \text{down}(p)$ as a potential translation of policy p . Why? If any $d \in D$ implements p , then the lemma tells us that $p = \text{up}(p')$. On the other hand, if $p \neq \text{up}(p')$, then no $d \in D$ can implement p .

A second standard lemma about Galois connections lends further insight.

Lemma 5.18 (reversal) *If $\text{down} : U \rightarrow D$ and $\text{up} : D \rightarrow U$ are monotone, then (down, up) is a Galois connection if and only if for all u , $u \leq \text{up}(\text{down}(u))$ and for all d , $\text{down}(\text{up}(d)) \leq d$.*

Applied to policy translation, this lemma tells us that $p \subseteq \text{up}(\text{down}(p))$, that is that $\text{up}(\text{down}(p))$ is *live*. In policy translation, we also want *safety*, that is, $\text{up}(\text{down}(p)) \subseteq p$.

5.2.1 Direct Policy Translation and Policy Spaces

Using Theorem A.9, we can extend the Galois connection between policies on R and policies on R' to a Galois connection between policy spaces on R and policy spaces on R' .

$$\begin{aligned} \text{down}^* &: \mathcal{P}(\mathcal{P}(R)) \rightarrow \mathcal{P}(\mathcal{P}(R')) \\ \text{down}^*(ps) &= \{\text{down}(p) \mid p \in ps\} \\ \text{up}^* &: \mathcal{P}(\mathcal{P}(R')) \rightarrow \mathcal{P}(\mathcal{P}(R)) \\ \text{up}^*(ps') &= \{\text{up}(p') \mid p' \in ps'\} \end{aligned}$$

The function up^* preserves indistinguishability, as the following lemma shows.

Lemma 5.19 *Let ps' be a policy space on R' and let $ps = \text{up}^*(ps')$. Then, for all r_1 and r_2 , $r_1 \equiv_{ps} r_2$ if and only if $f(r_1) \equiv_{ps'} f(r_2)$.*

The analogous lemma for down^* does not hold. That is, there are request sets R , R' , a function $f : R \rightarrow R'$, and a policy space ps on R with $ps' = \text{down}^*(ps)$ and requests $r_1, r_2 \in R$ such that $r_1 \equiv_{ps} r_2$ but $f(r_1) \not\equiv_{ps'} f(r_2)$. There are also requests r_1 and r_3 such that $f(r_1) \equiv_{ps'} f(r_3)$ but $r_1 \not\equiv_{ps} r_3$. Here is an example.

$$\begin{aligned}
R &= \{r_1, r_2, r_3\} \\
R' &= \{r'_1, r'_2\} \\
ps &= \{\{r_3\}\} \\
f(r_1) &= f(r_3) = r'_1 \\
f(r_2) &= r'_2
\end{aligned}$$

By construction,

$$ps' = \text{down}^*(ps) = \text{down}^*(\{\{r_3\}\}) = \{\text{down}(\{r_3\})\} = \{\{f(r_3)\}\} = \{\{r'_1\}\}$$

Observe that $r_1 \equiv_{ps} r_2$ but that $f(r_1) = r'_1 \not\equiv_{ps'} r'_2 = f(r_2)$. Also observe that $f(r_1) = r'_1 \equiv_{ps'} r'_1 = f(r_3)$ but that $r_1 \not\equiv_{ps} r_3$.

Fortunately, we can use Lemma 5.19 to prove that the image of an equivalence class under up^* is another equivalence class.

Lemma 5.20 *Let ps' be a policy space on R' and let $ps = up^*(ps')$. If c' is an equivalence class of $\equiv_{ps'}$, then $up^*(c') = \{\}$ or $up^*(c')$ is an equivalence class of \equiv_{ps} .*

We can also prove that up^* preserves additivity and completeness.

Lemma 5.21 *If ps' is additive then $up^*(ps')$ is additive.*

Lemma 5.22 *If ps' is complete then $up^*(ps')$ is complete.*

5.3 Composite Access Control Systems

In SPiCE, a variety of low-level enforcement mechanisms work together as a composite access control system. To model this, we assume that there is some set of enforcers E , and that each $e \in E$ can be modeled as an access control system with request set R_e . A request to the composite system involves making a request of some, but not necessarily all, of the enforcers. So, we model a request to the composite system as a product of enforcer requests, one for each enforcer, using the special token `none` to indicate that an enforcer is not involved in a request.

$$R = \prod_{e \in E} (R_e \cup \{\text{none}\})$$

Another choice would be to allow multiple requests for each enforcer, but we have found using a single request sufficient to model SPiCE.

Now that we have defined requests for a composite system, our framework automatically gives us a notion of policy, namely, $\mathcal{P}(R)$. However, each enforcer $e \in E$ is also equipped with its individual notion of policy $\mathcal{P}(R_e)$. To relate these notions, we define a *configuration* of enforcers as a product of enforcer policies.

$$\text{Config} = \prod_{e \in E} \mathcal{P}(R_e)$$

The name “configuration” is intended to be evocative; it corresponds to populating all the enforcers with their specific kind of configuration information (domain maps, type maps, packet filters, ...).

We relate policies on the composite system to configurations of enforcers with a pair of functions:

$$\begin{aligned}
down &: \mathcal{P}(R) \rightarrow Config \\
down(p) &= \prod_{e \in E} (\{r.e \mid r \in p\} - \{\text{none}\}) \\
up &: Config \rightarrow \mathcal{P}(R) \\
up(c) &= \{r \mid \forall e \in E. r.e \in (c.e \cup \{\text{none}\})\}
\end{aligned}$$

We write $r.e$ to denote extraction of the e 'th component of r . Unsurprisingly, $(down, up)$ is a Galois connection. To see this, for each e , apply Theorem A.11 to construct a Galois connection between $\mathcal{P}(R_e \cup \{\text{none}\})$ and $\mathcal{P}(R_e)$. Then, apply Theorem A.7 to construct a Galois connection between $\prod_{e \in E} \mathcal{P}(R_e \cup \{\text{none}\})$ and $\prod_{e \in E} \mathcal{P}(R_e)$. Next, apply Theorem A.20 to construct a Galois connection between $\mathcal{P}(\prod_{e \in E} (R_e \cup \{\text{none}\}))$ and $\prod_{e \in E} \mathcal{P}(R_e \cup \{\text{none}\})$. Finally, use Theorem A.8 to compose these Galois connections, yielding a Galois connection between $\mathcal{P}(\prod_{e \in E} (R_e \cup \{\text{none}\}))$ and $\prod_{e \in E} \mathcal{P}(R_e)$, that is, between $\mathcal{P}(R)$ and $Config$. Summarizing,

$$\begin{aligned}
&\mathcal{P}(R) \\
&= \mathcal{P}(\prod_{e \in E} (R_e \cup \{\text{none}\})) \\
&\rightleftharpoons \prod_{e \in E} \mathcal{P}(R_e \cup \{\text{none}\}) \\
&\rightleftharpoons \prod_{e \in E} \mathcal{P}(R_e) \\
&= Config
\end{aligned}$$

We use $U \rightleftharpoons D$ to indicate a Galois connection between U and D . This picture shows that there are two approximations going on. The first reflects the fact that the enforcers handle their requests independently, while the second reflects the fact that enforcers can not affect high-level requests that do not ask anything of them. For example, consider the situation with two enforcers described below.

$$\begin{aligned}
E &= \{e_1, e_2\} \\
R_{e_1} &= \{a_1, a_2\} \\
R_{e_2} &= \{b\} \\
p &= \{(a_1, b), (a_2, \text{none})\} \\
down(p) &= (\{a_1, a_2\}, \{b\}) \\
up(\{a_1, a_2\}, \{b\}) &= p \cup \{(a_1, \text{none}), (a_2, b)\}
\end{aligned}$$

In this example, a high-level policy, p , that accepts two composite requests is mapped down to a low-level configuration, $(\{a_1, a_2\}, \{b\})$, which is then mapped back up to a high-level policy. Because $(down, up)$ is a Galois connection, we know that $p \subseteq up(down(p))$. The additional requests correspond to the two approximations that this kind of policy makes.

5.4 Modeling Abstraction

Suppose we want to translate a high-level access control system with requests from R into a low-level access control system with requests from R' . In translating a high-level policy, a single authorized high-level request may authorize a number of low-level requests. This could happen because the high-level request is abstract; for example, it could include a role that represents many users. It could also happen because the high-level request implicitly allows dynamism that becomes explicit in the low-level request; for example, it may allow a user to connect from any of a number of machines. Fortunately, both of these situations can be modeled in the same way, as a function that maps a single high-level request into a set of low-level requests.

$$split : R \rightarrow \mathcal{P}(R')$$

As we did in Section 5.2, we explain the intended meaning of *split* by supposing that we have some policy p' for the low-level system and asking ourselves what policy in the high-level system p' represents. Consider a high-level request r . The situation is more complicated than in Section 5.2 because of abstraction; however, one of three things must be true. Either $split(r) \subseteq p'$, in which case r is *always* accepted, or $split(r) \cap p' = \{\}$, in which case r is *never* accepted, or neither of those is true, in which case r is *sometimes* accepted.

An access control system in which a request is sometimes accepted would be confusing indeed, so we define the meaning of a low-level policy to be the set of high-level requests that it always accepts.

$$\begin{aligned} up : \mathcal{P}(R') &\rightarrow \mathcal{P}(R) \\ up(p') &= \{r \mid split(r) \subseteq p'\} \end{aligned}$$

Suppose that we would like to translate a policy p on R into a policy p' on R' . We model the translator as a function

$$down : \mathcal{P}(R) \rightarrow \mathcal{P}(R')$$

What property would we like $down(p)$ to satisfy? To ensure that it accepts enough requests, if $r \in p$ and $r' \in split(r)$, then we would like to have $r' \in down(p)$. This is easily achieved by defining $down$ as follows.

$$down(p) = \bigcup_{r \in p} split(r)$$

From Corollary A.19, we know that $(down, up)$ is a Galois connection. Nevertheless, we must be careful; this Galois connection does not capture our intent. Consider the following situation.

$$\begin{aligned} R &= \{r_1, r_2\} \\ R' &= \{r'_1, r'_2\} \\ split(r_1) &= \{r'_1, r'_2\} \\ split(r_2) &= \{r'_2, r'_3\} \\ p &= \{r_1\} \\ down(p) &= down(\{r_1\}) = split(r_1) = \{r'_1, r'_2\} \\ up(down(p)) &= up(\{r'_1, r'_2\}) = \{r_1\} = p \end{aligned}$$

As the computation above shows, $p = up(down(p))$. So, we might expect that $down(p)$ is a reasonable translation of p . However, it is not – the problem is that $down(p)$ accepts r_2 *some of the time*, when it *splits* to r'_2 . This is wrong because the high-level policy p does not allow r_2 . One obvious fix might be to change the definition of $up(p')$ from $\{r \mid split(r) \subseteq p'\}$ to $\{r \mid split(r) \cap p' \neq \{\}\}$. But, that would just shift the problem the other way. We could find a policy p such that $p = up(down(p))$ and yet some requests in p would be rejected some of the time. To correctly capture our intention that the low-level policy either accept each request all of the time or none of the time, we need to change the low-level policy space.

Define a policy p' on R' to be consistent if for all $r \in R$, if $split(r) \cap p' \neq \{\}$ then $split(r) \subseteq p'$.

Intuitively, a policy p' is consistent if for every request, it either always accepts the request or always rejects the request. To avoid the problems we saw above, we would like our translation to produce a consistent policy. Fortunately, the following theorem lets us refine an existing Galois connection by an appropriate notion of consistency.

Theorem 5.23 *Suppose U and D are partial orders, (down, up) is a Galois connection between U and D , and D' is a subset of D that is closed under \sqcap . Define*

$$\begin{aligned} \text{close} &: D \rightarrow D' \\ \text{close}(d) &= \sqcap\{d' \mid d \leq d' \text{ and } d' \in D'\} \\ \text{down}' &: U \rightarrow D' \\ \text{down}'(u) &= \text{close}(\text{down}(u)) \\ \text{up}' &: D' \rightarrow U \\ \text{up}'(d') &= \text{up}(d') \end{aligned}$$

Then, $(\text{down}', \text{up}')$ is a Galois connection between U and D' .

To apply this to the Galois connection between $\mathcal{P}(R)$ and $\mathcal{P}(R')$ based on *split*, define $\text{Consis} = \{p' \in \mathcal{P}(R') \mid p' \text{ is consistent}\}$. We then need the following lemma.

Lemma 5.24 *The consistent policies in $\mathcal{P}(R')$ are closed under intersection.*

With that in place, Theorem 5.23 gives us the following Galois connection.

$$\begin{aligned} \text{close} &: \mathcal{P}(R') \rightarrow \text{Consis} \\ \text{close}(p') &= \cap\{p'_1 \mid p' \subseteq p'_1 \text{ and } p'_1 \text{ is consistent}\} \\ \text{down}' &: \mathcal{P}(R) \rightarrow \text{Consis} \\ \text{down}'(p) &= \text{close}(\text{down}(p)) = \text{close}(\bigcup_{r \in p} \text{split}(r)) \\ \text{up}' &: \text{Consis} \rightarrow \mathcal{P}(R) \\ \text{up}'(p') &= \text{up}(p') = \{r \mid \text{split}(r) \subseteq p'\} \end{aligned}$$

This Galois connection avoids the problem with requests being accepted some of the time. Because it only deals with consistent low-level policies, we could have equivalently defined up' by:

$$\text{up}'(p') = \{r \mid \text{split}(r) \cap p' \neq \{\}\}$$

The new Galois connection also clarifies what went wrong in our earlier counterexample. Now, $\{r'_1, r'_2\}$ is not consistent, and we have:

$$\begin{aligned} \text{down}'(\{r_1\}) &= \text{close}(\text{split}(r_1)) = \text{close}(\{r'_1, r'_2\}) = \{r'_1, r'_2, r'_3\} \\ \text{up}'(\text{down}'(\{r_1\})) &= \text{up}'(\{r'_1, r'_2, r'_3\}) = \{r_1, r_2\} \end{aligned}$$

This makes it clear for policy $p = \{r_1\}$ that $p \neq \text{up}'(\text{down}'(p))$.

5.5 Modeling SPiCE

To model the policy translation of SPiCE, we use our techniques for abstraction and composite systems. At the high level, we have Cape requests, R_{Cape} . At the low level, we have a family of enforcers E , with requests $R_l = \prod_{e \in E} (R_e \cup \{\text{none}\})$. Cape requests are abstractions of low-level

requests in that a Cape role can correspond to many users and a Cape resource can correspond to many resources. We model this with a function $split$ that takes a Cape request to a set of low-level requests.

$$split : R_{Cape} \rightarrow \mathcal{P}(R_l)$$

We also use $split$ to model the movement of low-level users from machine to machine by including in $split(r)$ requests originating from all the machines where a user taking on the role of r could be sitting. As in Section 5.4, $split$ leads to the following Galois connection.

$$\begin{aligned} down_1 &: \mathcal{P}(R_{Cape}) \rightarrow \mathcal{P}(R_l) \\ down_1(p) &= \bigcup_{r \in p} split(r) \\ up_1 &: \mathcal{P}(R_l) \rightarrow \mathcal{P}(R_{Cape}) \\ up_1(p_l) &= \{r \mid split(r) \subseteq p_l\} \end{aligned}$$

As in Section 5.3, we have low-level configurations $Config = \prod_{e \in E} \mathcal{P}(R_e)$ and the following Galois connection.

$$\begin{aligned} down_2 &: \mathcal{P}(R_l) \rightarrow Config \\ down_2(p_l) &= \prod_{e \in E} (\{r.e \mid r_l \in p_l\} - \{\text{none}\}) \\ up_2 &: Config \rightarrow \mathcal{P}(R_l) \\ up_2(c) &= \{r_l \mid \forall e \in E. r_l.e \in (c.e \cup \{\text{none}\})\} \end{aligned}$$

We compose these Galois connections to build a Galois connection between $\mathcal{P}(R_{Cape})$ and $Config$.

$$\begin{aligned} down &: \mathcal{P}(R_{Cape}) \rightarrow Config \\ down(p) &= \prod_{e \in E} (\bigcup_{r \in p} \bigcup_{r_l \in split(r)} r_l.e - \{\text{none}\}) \\ up &: Config \rightarrow \mathcal{P}(R_{Cape}) \\ up(c) &= \{r \mid \forall r_l \in split(r). \forall e \in E. r_l.e \in (c.e \cup \{\text{none}\})\} \end{aligned}$$

This Galois connection is almost the one we want. All that remains is to address the problem, which we already saw in Section 5.4, of configurations accepting a Cape request some, but not all, of the time. The solution we used in Section 5.4 works here. Define a configuration c to be consistent if for all $r \in R_{Cape}$, whenever $split(r) \cap up_2(c) \neq \{\}$ then $split(r) \subseteq up_2(c)$. Intuitively, a configuration is consistent if for every Cape request r , it either always accepts r or always rejects r .

We would like to apply Theorem 5.23 to restrict our Galois connection ($down, up$) to only deal with consistent configurations. Define $Consis = \{c \in Config \mid c \text{ is consistent}\}$. As our notion of greatest lower bound on configurations define:

$$\sqcap \{c_i \mid i \in I\} = \prod_{e \in E} (\bigcap_{i \in I} c_i.e)$$

We need the following lemma.

Lemma 5.25 *The set of consistent configurations is closed under greatest lower bound.*

This gives us a notion of closure on configurations.

$$\begin{aligned} close &: Config \rightarrow Consis \\ close(c) &= \sqcap \{c' \mid c \leq c' \text{ and } c' \text{ is consistent}\} \end{aligned}$$

With that in place, Theorem 5.23 gives us the following Galois connection.

$$\begin{aligned}
& \text{down}' : \mathcal{P}(R_{\text{Cape}}) \rightarrow \text{Consis} \\
& \text{down}'(p) = \text{close}(\prod_{e \in E} (\bigcup_{r \in p} \bigcup_{r_l \in \text{split}(r)} r_l.e - \{\text{none}\})) \\
& \text{up}' : \text{Consis} \rightarrow \mathcal{P}(R_{\text{Cape}}) \\
& \text{up}'(c) = \{r \mid \forall r_l \in \text{split}(r). \forall e \in E. r_l.e \in (c.e \cup \{\text{none}\})\}
\end{aligned}$$

In essence, the SPiCE compiler takes as input a Cape policy p and computes a candidate configuration $c = \text{down}'(p)$. It then checks if $p = \text{up}'(\text{close}(c))$. If they are equal, then SPiCE knows that c is a safe configuration of the enforcers. If not, then SPiCE can report an error message along with the additional requests in $\text{up}'(\text{close}(c)) - p$, assured by the representability lemma that there is no consistent c that implements the input Cape policy.

6 Related Work

The SPiCE translation system enables the first implementation of any of the CBAC access control models, and, to our knowledge, the first implementation of any coalition-based access control system. SPiCE is also the first system to translate abstract access control policies onto heterogeneous enforcer targets. There has been, however, a significant amount of work in the area of network policy synthesis, beyond the Smart Firewalls system that is targeted in our prototype. We also mention here an interesting paper on enforcer synthesis, along with pointers to OMG and IETF specifications for composing access policy enforcement mechanisms.

6.1 Network Policy Managers

A significant amount of research has been done in the area of network policy management in addition to the Smart Firewalls system. A few interesting recent works include the following.

[HKL03] describes a federated policy management system for managing homogeneous enforcers in telecommunications networks. The Houdini language is used both for specifying the intended access control policies and for configuring the enforcement mechanisms. Thus, the specified global policy is “decomposed” into local configurations for the enforcers, rather than “translated” into lower-level configuration languages. The authors provide formal semantics for both the global and local rulesets and prove the correctness of their decomposition algorithms.

[Kan01] considers the translation of high-level policies to low-level configurations for networks of homogeneous enforcers. In addition to the notion of “division” in which high-level policy is translated into one or more configurations for enforcers, Kanada considers the notion of “fusion” in which two or more high-level policies are transformed into a single configuration. While noting that certain elements of a high-level policy may introduce complexity that precludes fusion, this paper does not precisely detail the barriers to implementation.

While [ASH04] does not translate policies, the authors present techniques and algorithms for analyzing firewall policies, detecting anomalies, and editing existing policies. State diagrams provide the formal basis for analysis. The authors present a taxonomy of firewall policy anomalies which their policy advisor tool can detect and assist in correcting.

6.2 Other Policy Translators

[BDL03] describes an approach to automatically generating security architectures for distributed applications by first integrating security models with UML process models and then automatically generating executable systems from those integrated models. The authors define the SecureUML security modeling language, which generalizes RBAC, for writing the access control requirements.

This approach is quite different from that of SPiCE. SPiCE assumes existing enforcers with existing configuration semantics for access control. According to our formal semantics for Cape, OO-DTE's Sveltdt, and Smart Firewall's configurations, the SPiCE compiler preserves the Cape semantics on translation to the target configurations. [BDL03] effectively takes an RBAC policy and generates a custom enforcer, embedded in the specified application. The safety of the resulting configured enforcers (i.e., do they enforce the specified RBAC policy?) depends on whether there is a formal semantics for SecureUML and whether the code generator can be proven to respect that semantics in the translation process. One advantage of the [BDL03] approach is that the generated code is custom-tailored to enforce the given RBAC policy. In contrast, the SPiCE system must configure generic mechanisms to enforce the given Cape policy—sometimes this is not possible. One disadvantage of the [BDL03] approach is that it offers no assistance in configuring other enforcement mechanisms to support the application policy. Thus, while the generated application may accurately enforce the policy, an administrator must still coordinate the configurations of firewalls, databases, file systems, etc. to support the application-level enforcement.

6.3 Composite Access Control Systems

Our formal model views the collection of access policy enforcement mechanisms as a composite access control system in which individual enforcers mediate requests and render Boolean decisions which are then composed into an overall decision to allow or deny. Both the Object Management Group's (OMG's) CORBA Resource Access Decision (RAD) Facility and the Generic Authorization and Access control Application Program Interface (GAA-API) describe frameworks that support the composition of access control systems.

In the CORBA setting, the OMG's RAD Facility [Obj01] enables an application object to request an access control decision from an Access Decision object (ADO) which has a single method, `access_allowed`, that takes as input a `ResourceName`, an `Operation`, and `SecAttributes`, and returns a Boolean result. The product `ResourceName x Operation x SecAttributes` is analogous to a CBAC request, while the ADO captures a similar notion of policy. The ADO finds the `PolicyEvaluators` that are associated with a given resource, calls each evaluator to produce a Boolean result, then calls a `DecisionCombinator` to combine the results together. The RAD model is similar to ours, though less abstract and modular—requests are broken down and the model is not used recursively to express composite systems.

The GAA-API [RN00] defines a uniform authorization service interface for use by applications in requesting access control decisions. The `gaa_check_authorization` function is called by the application to generate an authorization decision. In this model, the results could be “yes” (operations authorized), “no” (access denied) or “maybe”, where “maybe” can mean that additional application-specific checks are required. This enables delegation of authority over access decisions through arbitrarily long chains of queries. However, the framework does not specify how termination of the mediation process can be ensured.

7 Discussion and Future Work

The SPiCE project has shown that automated configuration synthesis is feasible and that our formal results are applicable to authorization systems beyond the small suite implemented by our prototype. During our formal analysis, system design and prototype implementation, we identified a number of issues that impact the usability, scalability and deployability of policy translation systems. Many of these issues require solutions with both a formal or conceptual component and a systems design or implementation component. We discuss policy translation and dissemination issues for dynamic system environments, approaches to giving more control and feedback to the administrator, issues that must be resolved to make automated configuration synthesis practical, and experiments using SPiCE that might resolve some unanswered questions in computer security.

7.1 Dynamic System Environments

Before wireless connectivity and the advent of “self-healing” systems and networks, a “dynamic” system was one in which components failed, usually by crashing. Access control mechanisms were not required to respond to such failure events because crashed devices enabled no unintended access to their resources. Technology has changed: Laptops and PDAs arrive and depart from range of various wireless access points. Hosts crash and are automatically restarted. Networks partition and then reconnect to one another. Access control mechanisms must now adapt to changing distributed system conditions to ensure that access policies are enforced. Automated configuration synthesis is a necessity for environments in which system resources might spontaneously become more widely available than they were five minutes ago.

While the SPiCE system does not yet respond to changing conditions in the distributed system, we can identify research that must be accomplished to support dynamic configuration synthesis. The SPiCE policy compiler takes as one of its inputs a virtual system model, capturing the capabilities of the system’s access policy enforcers. Our prototype’s Network Oracle builds that virtual system model from static data stored in a database and from network topology data extracted from the Smart Firewalls policy engine. A necessary first step in building a dynamic configuration synthesis system is to receive status updates from a network management system, such as HP’s Openview, CA’s Unicenter, or Columbia University’s NESTOR. These systems rely on software agents running throughout the network to provide network health updates to a administrative console, identifying network connectivity changes, as well as responsiveness or unresponsiveness of hosts. The Network Oracle could be modified to take such status inputs, update its virtual system model and alert the SPiCE compiler to the change.

While enabling the SPiCE compiler to take virtual system model updates from the Network Oracle would require only a small engineering effort, determining what the compiler should do with those updates will require significant research into several apparently complex conceptual and systems problems. A reasonable response by the compiler to a new virtual system model would be recompilation of the current Cape policy. If the compilation fails, the compiler should provide helpful feedback to the administrator, identifying not only the cause of the problem (e.g., “the following accesses will be enabled, in addition to those permitted by the Cape policy”), but also recommending solutions (e.g., “install an OO-DTE web servlet enforcer on host 47 to protect files in /datafiles/images.”) Significant research into formal/conceptual foundations, heuristics, and systems analysis will be required to enable the SPiCE compiler to identify, analyze and select a

small set of solutions for presentation to the administrator.

If the compilation of a Cape policy for a new virtual system model is successful, then the Distributor can simply push the new configurations to the enforcers. However, if the changes to the virtual system model are small and localized, then we might wonder if a full recompilation is necessary. For configuration synthesis, what constitutes a “small” or “localized” change in the virtual system? Can we identify circumstances in which only a partial recompilation would be safe? Is there an optimal “granularity” for compilation, so that it would be more efficient to regenerate configurations for a larger portion of the distributed system than is strictly necessary to ensure safety? Formal/conceptual research into the safety of partial recompilation, as well as compiler/systems research into techniques for optimizing the translation will be necessary to answer many of these questions.

Suppose that we now have a new set of enforcer configurations, generated by either a full or partial recompilation of the Cape policy. The SPiCE Distributor must update the enforcement mechanisms with the new configuration. Must all the enforcers begin using these new configurations simultaneously to ensure safe policy enforcement? Of course, we can imagine scenarios where non-simultaneous configuration updates could introduce vulnerabilities. For example, suppose that under configuration suite C_1 , enforcer A is responsible for enforcing access policy on resource R, but under a new suite C_2 , enforcer B is mediating access to R. A vulnerability arises if A is informed by the receipt of its new configuration that it no longer needs to enforce access policy on R before B learns that it should now begin mediating access to R. Access requests for R will not be mediated until B receives its new configuration. While simultaneous configuration updates would be sufficient, assuming necessary atomicity requirements on request mediation are met to avoid requests that are “half way” through mediation under the old configuration when the configuration is received, there may be weaker requirements sufficient to ensure safety properties of the access control system. In distributed database environments, various notions of serializability are sufficient to ensure the consistency of replicas. An analogue to serializability must be defined for access configuration updates to ensure system safety. Of course, timeliness of updates is important, too, not just for progress or liveness requirements, but for system safety. Conceptual/formal and systems research are both required to define sufficient conditions to meet safety and liveness requirements for configuration updates.

While we have been focused on ensuring that the configurations generated for the access policy enforcers are safe, the enforcers need to ensure for themselves that the configurations they receive are correct. Many policy enforcers today may check the source and integrity of configurations they receive from administrative consoles by verifying the digital signature on a signed configuration object. An alternative may be feasible that would allow arbitrary entities to generate configurations from Cape policies and provide those configurations to subscribers. From the recent research in proof-carrying code [Nec97], we could adopt proof-carrying configurations that demonstrate that they implement the specified Cape policy for a given virtual system model. Small policy validators could be embedded in enforcers, enabling them to check each configuration update as it is received. Research will be required to determine how best to adapt the proof-carrying code results within the policy enforcement environment in which configurations are the “code.”

7.2 Giving More Control to the Administrator

The prototype SPiCE system automates the process of configuration synthesis: the administrator selects a policy to compile, clicks a “compile” button in the system control GUI and, if feasible, a configuration is generated, ready for deployment to the access policy enforcers. The administrator has no control over how the responsibility for enforcing a Cape policy is distributed over the enforcers. Currently, the SPiCE compiler requires each available enforcer to enforce as much of the Cape policy as it can, whether or not this results in redundant access mediation. In some cases, this approach will appropriately provide “defense in depth,” but, in other cases, may be too inefficient or redundant. As the capabilities of the compiler expand to handle a variety of enforcement mechanisms beyond our three targets, our system must choose which mechanisms will be used to enforce each element of the Cape policy for a portion of the distributed system. Enabling an administrator to state high-level preferences, such as “defense in depth” or “mediate once” or “highest assurance-rating first” would provide administrative control over the compiler’s approach to configuration synthesis.

In addition to offering administrative control through “preferences,” we may also offer the ability to specify the (partial) configuration of individual enforcers via “constraints”. For example, an administrator could add a constraint that requires certain ports on a firewall to remain closed. The SPiCE compiler would incorporate the constraints into the automatically generated configuration. Constraints are essential to integrating automated policy translation into environments where legacy systems exist, and external systems or rules restrict how enforcers may be configured.

Feedback from the SPiCE system to the administrator is critical to ensuring that Cape policies are correctly implemented by access policy enforcers. In Section 7.1, we discussed extending the SPiCE system to provide feedback on compilation failures and recommendations on solutions. Policy errors are another problem for which feedback from SPiCE (in this case, the Policy Editor, rather than the compiler) would be helpful. When user *Alice* explains to her local system administrator that she needs access to resource *F*, but the system is denying her requests, the administrator needs to know what, if anything, is wrong. Does the policy permit access or are there are failures in the distributed system that are preventing *Alice*’s access? If the policy currently denies access and there is reason to believe that *Alice* should have access, then it would be helpful if the Policy Editor could identify the “least” policy change necessary to grant the access. By *least*, we mean the lowest possible element of a Cape refinement path. In general, developing the capability to respond to queries about authorizations for access to specific resources would greatly assist administrators in understanding the practical implications of high-level policies and troubleshooting access problems within complex distributed systems.

While an access control systems are traditionally tasked with preventing accesses that violate a policy (a safety property), automatic configuration synthesis may help to expand the role of access control systems to include enabling access that is permitted according to a policy (a liveness property). Guaranteeing that a user can access all of the resources that a policy permits is well beyond our capabilities today. Note that this guarantee is much harder than “not impeding” access to a resource—the operational semantics of “allow” for current access policy enforcement mechanisms. With only a manual configuration process, access control systems cannot respond to distributed system changes. With automated configuration synthesis, it may be feasible for access control systems to interact with the “self-healing” facilities of advanced distributed systems to ensure the ability of users to access their resources. Research into the intersection of fault tolerant,

self-healing systems with access control systems may yield hybrids that help administrators ensure that users get exactly the access intended, no more and no less.

7.3 Making Configuration Synthesis Practical

The SPiCE system is a research prototype that proves the feasibility of the automated configuration synthesis approach. Many research and engineering issues must be addressed before a configuration synthesis system like SPiCE will be ready for deployment into real systems. We discuss some of the issues with significant research components below.

Distributed systems rely on a broad set of access policy enforcers, including file systems, database management systems, and desktop firewalls, in addition to the packet filtering firewalls, web servers and Java runtimes controlled by SPiCE. These systems also contain resources for which access policies refer to “how much” (e.g., network bandwidth, disk storage) in addition to the usual “allow” or “deny.” Resource allocation systems are used to reserve and monitor access to and consumption of such resources. Extending the SPiCE system to target a new type of enforcer will require addressing the formal characteristics of the new access control system, including any necessary extensions to our virtual system model, developing and verifying the correctness of translation algorithms and implementing the translation and communication between the Distributor and the enforcer. It is likely that for some types of enforcer, our translation approach will not be feasible. Only by developing formal semantics for the enforcer’s access control system will we be able to determine whether a SPiCE translation can meet our safety requirements. To extend the SPiCE system to express and enforce resource allocation policies, we must develop formal policy semantics and create safe translation algorithms. While we hope that the SPiCE approach can be extended to handle resource allocation policies, composing results from multiple such enforcers may be more difficult than composing Boolean responses from the simpler access policy enforcers that SPiCE currently supports.

While the SPiCE system handles some aspects of authority delegation through Cape policy refinement, concrete aspects of trust management in distributed systems must be addressed to make SPiCE practical to administer. For example, when the OO-DTE enforcers receive X.509 certificates, which credential issuers should they trust to assert the identity of the requesting principal? This question is currently answered via manually-built repositories of certificates representing trusted issuers. A natural extension of the SPiCE system would automatically generate trust policies in parallel with access configurations, fully automating the process of administering access policy enforcement mechanisms.

The SPiCE Policy Editor ensures policy refinement (delegation to successively lower-level entities and access to successively “smaller” resources), but currently provides no mechanism and no formal model for delegating authority for policy administration. Just as a role-based administrative model was developed for RBAC [SBM99], we may find that a coalition-based administrative model is appropriate for CBAC. Specifically, Cape policies are refined partly along the lines of organizational subordination. Thus, a policy statement for a suborganization will refine a broader policy statement for its parent organization. A coalition-based administrative model might designate a special role (e.g., `administrator`) that is **ActiveWithin** each `ORGANIZATION`. Default authorizations may be generated for a user acting in the `administrator` role for her `ORGANIZATION` to `modify the Cape policy resource`. Developing the semantics of such a model and implementing it in the SPiCE system will involve some formal/conceptual work and a significant amount of en-

gineering to tie the administrative authorizations expressed by Cape to administrative capabilities for policy enforcers like operating systems, database management systems, and firewalls.

While introducing the SPiCE system into a distributed system with no legacy enforcer configurations to uphold would be ideal, we do not expect the deployment process to be so easy. If legacy rules must continue to be enforced in support of an intra-organizational access policy, then ensuring that SPiCE generates coalition-focused configurations that co-exist peacefully with the legacy intra-organizational rules may be a challenge. As part of its safety check on potential configurations, the SPiCE compiler determines whether a configuration rule enabling an access will inadvertently enable accesses that are not permitted by Cape. It is possible that we will be able to extend the SPiCE safety checks to review legacy enforcer configurations and alert the administrator to potentially problematic consequences. Problems are likely to arise in this process. For example, name space clashes are likely to occur between the CBAC-based naming used in the SPiCE-generated configuration rules and the legacy name spaces of the existing enforcer rules. A principal may be inadvertently assigned two OO-DTE domains, causing the OO-DTE enforcer to generate an error. Similar, but potentially more dangerous clashes may occur if network names clash, allowing firewall rules (which are evaluated in sequence) to first deny then begin allowing communication. Research is needed to determine how much usable information can be provided to an administrator deploying SPiCE into an existing distributed system.

7.4 Experiments with SPiCE

We undertook the SPiCE project because we believed that replacing a manual security administration process with an automated process could reduce errors and vulnerabilities and improve the scalability of security administration in large distributed systems and coalition environments. Now that we have a simple prototype, we can begin designing preliminary tests for those hypotheses. However, we expect that serious testing must wait until our prototype is extended to support a broader range of “real” access policy enforcers, to ensure a fair comparison between the manual and automatic approaches.

Debate over the value of “defense-in-depth” has continued for years in the Information Assurance community. While some of the issues are conceptual (i.e., what constitutes “depth,” and how can we be sure that our enforcers are independent of one another), a significant practical problem has been the great difficulty in generating alternative enforcer configurations for testing. By extending SPiCE as described in Section 7.2 to support administrator preferences for distributing the responsibility for enforcing a Cape policy over various types of enforcers we will have a practical tool for testing theories about the assurance provided by layered defenses.

8 Conclusion

The SPiCE translation system simplifies the process of access policy construction and implementation by providing two tools to assist policy makers and administrators in creating and deploying access control policies: The Policy Editor helps policy makers write and correctly refine enterprise-level access control policies, while the SPiCE compiler automatically synthesizes implementations of those policies for the target distributed system. We have developed a formal framework to capture the semantics of our high-level and target policies and to prove the correctness and completeness of the SPiCE translation system.

The Policy Editor enables policy makers to express enterprise-level coalition access control policies and refine them to successively lower-levels of abstraction. Because the Policy Editor draws its data from the CBAC domain specification, it can prevent administrative user errors due to domain mismatch (e.g., an authorization written for an undefined role) or to incorrect refinement of abstract policy statements (e.g., a policy statement for low-level authorization does not refine a policy statement for a high-level organization because the low-level organization is not a child of the high-level organization.)

Given a high-level coalition access control policy, the SPiCE compiler automatically synthesizes configurations for Smart Firewalls and middleware-level OO-DTE access policy enforcers. If the given policy cannot be implemented with the available enforcers, the SPiCE compiler warns the administrator. We have developed formal semantics of all three access control systems (CBAC/Cape, OO-DTE/Sveldt, and Smart Firewalls) and proved that our translation algorithms respect those semantics (safety) and fail only when no semantically correct translation is possible (completeness).

In addition to proving the safety and completeness of our translation, our formal results document the characteristics of access control systems that can be modeled in our semantic framework. Thus, after building a semantic model of a new access control system, we can determine whether our translation approach can be used to generate configurations for that system. In addition, our framework treats source systems (e.g., CBAC/Cape) and target systems (e.g., Smart Firewalls) identically. Thus, we can model a composite access control system in which a high-level policy is translated to configurations for an access policy enforcement mechanism, which then further translate to even lower-level configurations for other access policy enforcers. In fact, this is exactly how our semantic framework handles the Cape \rightarrow Smart Firewalls \rightarrow Linux iptables translation.

References

- [ASH04] E. S. Al-Shaer and H. H. Hamed. Modeling and management of firewall policies. *IEEE eTransactions on Network and Service Management*, 1(1), April 2004.
- [BCG⁺01] J. Burns, A. Cheng, P. Gurung, S. Rajagopalan, P. Rao, D. Rosenbluth, A. V. Suren-dran, and D. M. Martin Jr. Automatic management of network security policy. In *Proceedings of the DARPA Information Survivability Conference and Exposition II (DARPA '01)*, volume 2, pages 12–26, Anaheim, CA, June 2001.
- [BDL03] D. Basin, J. Doser, and T. Lodderstedt. Model driven security for process-oriented systems. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, pages 100–109, Como, Italy, June 2003.
- [Bir40] G. Birkhoff. *Lattice Theory (first edition)*. American Mathematical Society, first edition, 1940.
- [BK85] W.E. Boebert and R.Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth National Computer Security Conference*, pages 18–27, Gaithersburg, MD, September 1985.
- [BSS⁺95] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghghat. Practical domain and type enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, Oakland, California, May 1995.

- [CTWS02] E. Cohen, R. K. Thomas, W. Winsborough, and D. Shands. Models for coalition-based access control (CBAC) [Extended Abstract]. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, pages 97–106, Monterey, California, June 2002.
- [DDLS01] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proceedings of Policy 2001: Workshop on Policies for Distributed Systems and Networks*, number 1995 in LNCS, pages 18–39, Bristol, UK, January 2001. Springer-Verlag.
- [FEM00] FEMA. FEMA Strategic Plan FY 2000—FY 2006. In http://www.fema.gov/library/splan_01.htm/, 2000.
- [FEM01] FEMA. FEMA website. In <http://www.fema.gov/>, 2001.
- [HKL03] R. Hull, B. Kumar, and D. Lieuwen. Towards federated policy management. In *Proceedings of the IEEE Fourth International Workshop on Policies for Distributed Systems and Networks (POLICY '03)*, Lake Como, Italy, June 2003.
- [Kan01] Y. Kanada. Policy division and fusion: Examples and a method—Or, multiple classifiers considered harmful. In *Proceedings of the Seventh IFIP/IEEE International Symposium on Integrated Network Management (IM 2001)*, pages 545–560, May 2001.
- [KYBR99] A. V. Konstantinou, Y. Yemini, S. Bhatt, and S. Rajagopalan. Managing security in dynamic networks. In *Proceedings of LISA '99: Thirteenth Systems Administration Conference*, pages 109–121, Seattle, WA, November 1999. USENIX.
- [Nec97] G. C. Necula. Proof-carrying code. In *Proceedings of the Symposium on Principles of Programming Languages*, Paris, France, January 1997.
- [Obj01] Object Management Group. Resource access decision facility specification, April 2001.
- [RN00] T. Ryutov and B. C. Neuman. Access control framework for distributed applications. draft-ietf-cat-acc-cntrl-frmw-05.txt, November 2000.
- [SBM99] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information System Security*, 2(1):105–135, 1999.
- [SCFY96] R. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.
- [STM+99] D. F. Sterne, G. W. Tally, C. D. McDonell, D. L. Sherman, D. L. Sames, and P. X. Pasturel. Scalable access control for distributed object systems. In *Proceedings of the Eighth USENIX Security Symposium*, Washington, D. C., August 1999.
- [SYJS01] D. Shands, R. Yee, J. Jacobs, and E. J. Sebes. Secure virtual enclaves: Supporting coalition use of distributed application technologies. *ACM Transactions on Information and System Security*, 4(2):103–133, May 2001.

[TSM98] G. Tally, D. F. Sterne, and C. D. McDonell. Sigma project: DTEL++ language specification. Technical report, Trusted Information Systems, Inc., 1998.

A Review

A.1 Set Theory

We explain our notation and review some basic definitions of set theory.

- \wedge , \vee , \Rightarrow , and \Leftrightarrow represent logical conjunction, disjunction, implication, and equivalence, respectively.
- “ $\forall x \in X$ ” should be read “for each element x in the set X ”.
- “ $\exists x \in X$ ” should be read “there exists an element x of the set X ”.
- A dot “.” is a scoping notation and stands for a left bracket whose mate is as far to the right as is possible without altering the pairing of left and right brackets already present. Of the quantifiers and connectives described above, \forall and \exists have the smallest scope, followed by \wedge , then \vee , then \Rightarrow and then \Leftrightarrow .
- “ $Y \subseteq X$ ” says that the set Y is a subset of X , i.e., each member of Y is also a member of X .
- “ $X \cup Y$ ” denotes the set-theoretic union of X and Y .
- “ $X \cap Y$ ” denotes the set-theoretic intersection of X and Y .
- “ $X \times Y$ ” denotes the cartesian product of X and Y .
- If $\{X_i \mid i \in I\}$ is a family of sets, then $\prod_{i \in I} X_i$ denotes their product. Given an element x of the product, we write $x.i$ to denote the i 'th component of x .
- If $\{f_i : X_i \rightarrow Y_i \mid i \in I\}$ is a family of functions, then their product $\prod_{i \in I} f_i$ is the function $f : \prod_{i \in I} X_i \rightarrow \prod_{i \in I} Y_i$ defined by $f(x) = \prod_{i \in I} x.i$.
- $\mathcal{P}(X)$ denotes the power set of a set X , that is, the set having as its members all subsets of X . $Y \subseteq X$ is logically equivalent to $Y \in \mathcal{P}(X)$.
- $\{x \in X \mid Q(x)\}$ denotes that subset of X consisting of those elements x for which the predicate $Q(x)$ holds.
- If $f : A \rightarrow B$ and $g : B \rightarrow C$, then their *composition*, $(g \circ f) : A \rightarrow C$ is defined by $(g \circ f)(a) = g(f(a))$.
- If $f : A \rightarrow B$, then its *pointwise extension* $f^* : \mathcal{P}(A) \rightarrow \mathcal{P}(B)$ is defined by $f^*(as) = \{f(a) \mid a \in as\}$.

We include some definitions about relations:

- A function $\mathbf{F} : A \rightarrow B$ defines a functional relation $\mathbf{F} : A \times B$ where $\mathbf{F}(a) = b$ if and only if $\mathbf{F}(a, b)$.

- A binary relation \mathbf{R} on the set A is *reflexive* if and only if $\forall a \in A. R(a, a)$.
- A binary relation \mathbf{R} on the set A is *transitive* if and only if $\forall a, b, c \in A. R(a, b) \wedge R(b, c) \Rightarrow R(a, c)$.
- A binary relation \mathbf{R} on the set A is *antisymmetric* if and only if $\forall a, b \in A. R(a, b) \wedge R(b, a) \Rightarrow a = b$.
- A binary relation is an *equivalence relation* if and only if it is reflexive, transitive, and symmetric.
- A binary relation \mathbf{R} is a *partial order* on if and only if it is reflexive, transitive and antisymmetric.
- A binary relation \mathbf{R} is a *forest* on the set A if and only if \mathbf{R} imposes a partial order on A and $\forall a, b, c \in A, \mathbf{R}(a, b) \wedge \mathbf{R}(a, c) \Rightarrow \mathbf{R}(b, c) \vee \mathbf{R}(c, b)$.
- A relation $\mathbf{Q}: A \times B$ is an *extension* of a relation $\mathbf{R}: C \times D$ if and only if $C \subseteq A \wedge D \subseteq B \wedge \forall c \in C \forall d \in D, \mathbf{Q}(c, d) \Leftrightarrow \mathbf{R}(c, d)$.

We include some definitions and facts about partial orders:

- If $\{X_i \mid i \in I\}$ is a family of partially ordered sets, then their product $\prod_{i \in I} X_i$ is partially ordered by the pointwise order, $x \leq x' = \forall i \in I. x.i \leq x'.i$.
- If A and B are partial orders, and $f : A \rightarrow B$, then f is *monotone* if and only if for all $a_1, a_2 \in A$, if $a_1 \leq a_2$, then $f(a_1) \leq f(a_2)$.
- If A is a set, then \subseteq is a partial order on $\mathcal{P}(A)$ that we refer to as the *subset order*.
- If A is partially ordered, then we define the *powerset order* on $\mathcal{P}(A)$ by $as_1 \leq as_2$ if and only if for all a_1 in as_1 , there exists $a_2 \in as_2$ such that $a_1 \leq a_2$.
- If A is partially ordered, then u is an *upper bound* of a set $S \subseteq A$ if and only if for all $a \in S$, $a \leq u$.
- If A is partially ordered, then u is the *least upper bound* of a set $S \subseteq A$ if and only if whenever u' is an upper bound of S , then $u \leq u'$. We write $\sqcup S$ to denote the least upper bound of S .
- If A is partially ordered, then l is an *lower bound* of a set $S \subseteq A$ if and only if for all $a \in S$, $l \leq a$.
- If A is partially ordered, then l is the *greatest lower bound* of a set $S \subseteq A$ if and only if whenever l' is a lower bound of S , then $l' \leq l$. We write $\sqcap S$ to denote the greatest lower bound of S .
- If A is partially ordered, then A is a *complete lattice* if every subset of A has a greatest lower bound and a least upper bound.

A.2 Galois Connections

Galois connections were first introduced in 1940 [Bir40] to model the correspondence between subgroups of the Galois group and subfields of a separable field extension, which is the subject of the Fundamental Theorem of Galois Theory. They have since been used to formalize approximations in many contexts.

Definition A.1 (Galois connection) *Let U and D be partially ordered sets, $down : U \rightarrow D$, and $up : D \rightarrow U$. Then $(down, up)$ is a Galois connection between U and D if $down$ and up are monotone and if for all $u \in U$ and $d \in D$, $down(u) \leq d$ if and only if $u \leq up(d)$.*

Lemma A.2 (reversal) *If $down : U \rightarrow D$ and $up : D \rightarrow U$ are monotone, then $(down, up)$ is a Galois connection if and only if for all u , $u \leq up(down(u))$ and for all d , $down(up(d)) \leq d$.*

Lemma A.3 *If $(down, up)$ is a Galois connection, then $up(d) = \sqcup\{u \mid down(u) \leq d\}$.*

Lemma A.4 *If $(down, up)$ is a Galois connection, then $down(u) = \sqcap\{d \mid u \leq up(d)\}$.*

Lemma A.5 (representability) *If $(down, up)$ is a Galois connection between U and D , then $u = up(down(u))$ if and only if there exists $d \in D$ such that $u = up(d)$.*

Lemma A.6 (representability) *If $(down, up)$ is a Galois connection between U and D , then $d = down(up(d))$ if and only if there exists $u \in U$ such that $down(u) = d$.*

Theorem A.7 (product of Galois connections is a Galois connection) *Suppose $\{U_i \mid i \in I\}$ and $\{D_i \mid i \in I\}$ are families of partially ordered sets, and $\{(down_i : U_i \rightarrow D_i, up_i : D_i \rightarrow U_i) \mid i \in I\}$ is a family of Galois connections. Then $(\prod_{i \in I} down_i, \prod_{i \in I} up_i)$ is a Galois connection between $\prod_{i \in I} U_i$ and $\prod_{i \in I} D_i$, each under the pointwise order.*

Theorem A.8 (composition of Galois connections is a Galois connection) *If $(down_1, up_1)$ is a Galois connection between A and B , and $(down_2, up_2)$ is Galois connection between B and C , then $(down_2 \circ down_1, up_1 \circ up_2)$ is a Galois connection between A and C .*

Theorem A.9 (pointwise extension) *If $(down, up)$ is a Galois connection between U and D , then their pointwise extensions form a Galois connection between $\mathcal{P}(U)$ and $\mathcal{P}(D)$ under the powerset order.*

Lemma A.10 (pointwise implements) *If $(down, up)$ is a Galois connection between U and D and $down'$ and up' are the pointwise extensions of $down$ and up , then for all $us \in \mathcal{P}(U)$, there exists $ds \in \mathcal{P}(D)$ such that $us \subseteq up'(ds)$ if and only if for all $u \in us$, $u = up(down(u))$.*

Theorem A.11 *Suppose A and B are sets. Define*

$$\begin{aligned} down &: \mathcal{P}(A \cup B) \rightarrow \mathcal{P}(A) \\ down(abs) &= abs - B \\ up &: \mathcal{P}(A) \rightarrow \mathcal{P}(A \cup B) \\ up(as) &= as \cup B \end{aligned}$$

Then, $(down, up)$ is a Galois connection, with $\mathcal{P}(A \cup B)$ and $\mathcal{P}(A)$ under the subset order.

Theorem A.12 *Suppose A and B are sets. Define*

$$\begin{aligned} \text{down} &: \mathcal{P}(A \cup B) \rightarrow \mathcal{P}(A) \\ \text{down}(abs) &= abs \cap A \\ \text{up} &: \mathcal{P}(A) \rightarrow \mathcal{P}(A \cup B) \\ \text{up}(as) &= as \cup (B - A) \end{aligned}$$

Then, (down, up) is a Galois connection, with $\mathcal{P}(A \cup B)$ and $\mathcal{P}(A)$ under the subset order.

Theorem A.13 (function and inverse form Galois connection) *Suppose A and B are sets and $f : A \rightarrow B$. Define*

$$\begin{aligned} \text{down} &: \mathcal{P}(A) \rightarrow \mathcal{P}(B) \\ \text{down}(as) &= \{f(a) \mid a \in as\} \\ \text{up} &: \mathcal{P}(B) \rightarrow \mathcal{P}(A) \\ \text{up}(bs) &= \{a \mid f(a) \in bs\} \end{aligned}$$

Then (down, up) is a Galois connection, with $\mathcal{P}(A)$ and $\mathcal{P}(B)$ under the subset order.

Theorem A.14 (glb forms a Galois connection) *Suppose A is a complete lattice and I is a set. Define*

$$\begin{aligned} \text{down} &: A \rightarrow (I \rightarrow A) \\ \text{down}(a)(i) &= a \\ \text{up} &: (I \rightarrow A) \rightarrow A \\ \text{up}(f) &= \sqcap \{f(i) \mid i \in I\} \end{aligned}$$

Then, (down, up) is a Galois connection, with $I \rightarrow A$ ordered pointwise.

Corollary A.15 *Suppose U is a complete lattice, $\{D_i \mid i \in I\}$ is a family of partially ordered sets, and for all $i \in I$, $(\text{down}_i, \text{up}_i)$ is a Galois connection between U and D_i . Define*

$$\begin{aligned} \text{down} &: U \rightarrow \prod_{i \in I} D_i \\ \text{down}(u)(i) &= \text{down}_i(u) \\ \text{up} &: \prod_{i \in I} D_i \rightarrow U \\ \text{up}(g) &= \sqcap \{\text{up}_i(g(i)) \mid i \in I\} \end{aligned}$$

Then, (down, up) is a Galois connection, with $\prod_{i \in I} D_i$ ordered pointwise.

Theorem A.16 (lub forms Galois connection) *Suppose A is a complete lattice. Define*

$$\begin{aligned} \text{down} &: \mathcal{P}(A) \rightarrow A \\ \text{down}(as) &= \sqcup as \\ \text{up} &: A \rightarrow \mathcal{P}(A) \\ \text{up}(a) &= \{a' \mid a' \leq a\} \end{aligned}$$

Then, (down, up) is a Galois connection, with $\mathcal{P}(A)$ under the powerset order or under the subset order.

Corollary A.17 *Suppose A is a set. Define*

$$\begin{aligned}
\text{down} &: \mathcal{P}(\mathcal{P}(A)) \rightarrow \mathcal{P}(A) \\
\text{down}(s) &= \cup s \\
\text{up} &: \mathcal{P}(A) \rightarrow \mathcal{P}(\mathcal{P}(A)) \\
\text{up}(as) &= \mathcal{P}(as)
\end{aligned}$$

Then (down, up) is a Galois connection, with $\mathcal{P}(\mathcal{P}(A))$ and $\mathcal{P}(A)$ under the subset order.

Corollary A.18 Suppose A is a partial order, B is a complete lattice, and $f : A \rightarrow B$ is a monotone function. Define

$$\begin{aligned}
\text{down} &: \mathcal{P}(A) \rightarrow B \\
\text{down}(as) &= \sqcup \{f(a) \mid a \in as\} \\
\text{up} &: B \rightarrow \mathcal{P}(A) \\
\text{up}(b) &= \{a \mid f(a) \leq b\}
\end{aligned}$$

Then, (down, up) is a Galois connection, with $\mathcal{P}(A)$ under the powerset order.

Corollary A.19 Suppose A and B are sets, and $f : A \rightarrow \mathcal{P}(B)$. Define

$$\begin{aligned}
\text{down} &: \mathcal{P}(A) \rightarrow \mathcal{P}(B) \\
\text{down}(as) &= \cup \{f(a) \mid a \in as\} \\
\text{up} &: \mathcal{P}(B) \rightarrow \mathcal{P}(A) \\
\text{up}(bs) &= \{a \mid f(a) \subseteq bs\}
\end{aligned}$$

Then (down, up) is a Galois connection, with $\mathcal{P}(A)$ and $\mathcal{P}(B)$ under the subset order.

Theorem A.20 (approximate set of tuples by tuple of sets) Suppose $\{A_i \mid i \in I\}$ is a family of nonempty partially ordered sets. Define

$$\begin{aligned}
\text{down} &: \mathcal{P}(\prod_{i \in I} A_i) \rightarrow \prod_{i \in I} \mathcal{P}(A_i) \\
\text{down}(fs)(i) &= \{f(i) \mid f \in fs\} \\
\text{up} &: \prod_{i \in I} \mathcal{P}(A_i) \rightarrow \mathcal{P}(\prod_{i \in I} A_i) \\
\text{up}(g) &= \prod_{i \in I} g(i)
\end{aligned}$$

Then (down, up) is a Galois connection, with $\mathcal{P}(\prod_{i \in I} A_i)$ and $\mathcal{P}(A_i)$ under the powerset or subset order, and $\prod_{i \in I} \mathcal{P}(A_i)$ ordered pointwise.

Theorem A.21 (equivalence relation forms Galois connection) Suppose A is a set and \equiv is an equivalence relation on A . Let C be the set of equivalence classes of \equiv . Define

$$\begin{aligned}
\text{down} &: \mathcal{P}(A) \rightarrow \mathcal{P}(C) \\
\text{down}(as) &= \{c \mid c \cap as \neq \{\}\} \\
\text{up} &: \mathcal{P}(C) \rightarrow \mathcal{P}(A) \\
\text{up}(cs) &= \cup cs
\end{aligned}$$

Then (down, up) is a Galois connection, with $\mathcal{P}(A)$ and $\mathcal{P}(C)$ under the subset order.

Theorem A.22 Suppose A and B are sets, $f : A \rightarrow \mathcal{P}(B)$, and \equiv is an equivalence relation on A . Define

$$\begin{aligned}
\text{down} &: \mathcal{P}(A) \rightarrow \mathcal{P}(B) \\
\text{down}(as) &= \cup\{f(a') \mid \exists a \in as. a \equiv a'\} \\
\text{up} &: \mathcal{P}(B) \rightarrow \mathcal{P}(A) \\
\text{up}(bs) &= \{a \mid \forall a' \equiv a. f(a') \leq bs\}
\end{aligned}$$

Then (down, up) is a Galois connection.

Corollary A.23 Suppose A and B are sets, and $R : \mathcal{P}(A \times B)$. Define a binary relation R_A on A by $a_1 R_A a_2$ if there exists b such that $a_1 R b$ and $a_2 R b$. Let \equiv be the smallest equivalence relation containing R_A . Define

$$\begin{aligned}
\text{down} &: \mathcal{P}(A) \rightarrow \mathcal{P}(B) \\
\text{down}(as) &= \{b \mid \exists a \in as \text{ and } a' \equiv a. a' R b\} \\
\text{up} &: \mathcal{P}(B) \rightarrow \mathcal{P}(A) \\
\text{up}(bs) &= \{a \mid \forall a' \equiv a. \forall b \in B. a R b \Rightarrow b \in bs\}
\end{aligned}$$

Then (down, up) is a Galois connection.

Theorem A.24 Suppose U is a partial order, D is a complete lattice, and $\text{up} : D \rightarrow U$ is monotone. Define

$$\begin{aligned}
\text{down} &: U \rightarrow D \\
\text{down}(u) &= \sqcap\{d \mid u \leq \text{up}(d)\}
\end{aligned}$$

Then down is monotone.

The above construction does not necessarily lead to a Galois connection between U and D . Consider the following situation:

$$\begin{aligned}
U &= \{u_1, u_2, u_3\} \\
u_1 &\leq u_2 \leq u_3 \\
D &= \{d_1, d_{2a}, d_{2b}, d_3\} \\
d_1 &\leq d_{2a} \leq d_3 \\
d_1 &\leq d_{2b} \leq d_3 \\
\text{up}(d_1) &= u_1 \\
\text{up}(d_{2a}) &= \text{up}(d_{2b}) = u_2 \\
\text{up}(d_3) &= u_3
\end{aligned}$$

Then, U is a partial order, D is a complete lattice, and up is monotone. If $\text{down}(u) = \sqcap\{d \mid u \leq \text{up}(d)\}$, then

$$\begin{aligned}
\text{down}(u_1) &= \sqcap\{d_1, d_{2a}, d_{2b}, d_3\} = d_1 \\
\text{down}(u_2) &= \sqcap\{d_{2a}, d_{2b}, d_3\} = d_1 \\
\text{down}(u_3) &= \sqcap\{d_3\} = d_3
\end{aligned}$$

So, $\text{up}(\text{down}(u_2)) = \text{up}(d_1) = u_1$. So we do not have that $u_2 \leq \text{up}(\text{down}(u_2))$.

Theorem A.25 Suppose U and D are partial orders, (down, up) is a Galois connection between U and D , and D' is a subset of D that is closed under \sqcap . Define

$$\begin{aligned} \text{close} &: D \rightarrow D' \\ \text{close}(d) &= \sqcap \{d' \mid d \leq d' \text{ and } d' \in D'\} \\ \text{down}' &: U \rightarrow D' \\ \text{down}'(u) &= \text{close}(\text{down}(u)) \\ \text{up} &: D' \rightarrow U \\ \text{up}'(d') &= \text{up}(d') \end{aligned}$$

Then, $(\text{down}', \text{up}')$ is a Galois connection between U and D' . Also, close is monotone and for all $d' \in D'$, $d' \leq \text{close}(d')$.