January 2013

Toward More Composable Software-Security Policies: Tools and Techniques

Daniel Lomsak
University of South Florida, dlomsak@mail.usf.edu

Toward More Composable Software-Security Policies: Tools and Techniques

by

Daniel Lomsak

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Hao Zheng, Ph.D.
Yao Liu, Ph.D.
Nataša Jonoska, Ph.D.
Salvatore Morgera, Ph.D.

Date of Approval:
March 25, 2013

Keywords: Software Engineering, Visual Specification, Signed Regular Languages,
Policy-Specification Languages, Policy Composition

DEDICATION

To my wife, who believed in me before I believed in myself, and endured alongside me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Complex software-security policies are difficult to specify, understand, and update. The same is true for complex software in general, but while many tools and techniques exist for decomposing complex general software into simpler reusable modules (packages, classes, functions, aspects, etc.), few tools exist for decomposing complex security policies into simpler reusable modules. The tools that do exist for modularizing policies either encapsulate entire policies as atomic modules that cannot be decomposed or allow fine-grained policy modularization but require expertise to use correctly.

This dissertation presents a policy-composition tool called PoliSeer [27, 26] and the PoCo policy-composition software-security language. PoliSeer is a GUI-based tool designed to enable users who are not expert policy engineers to flexibly specify, visualize, modify, and enforce complex runtime policies on untrusted software. PoliSeer users rely on expert policy engineers to specify universally composable policy modules; PoliSeer users then build complex policies by composing those expert-written modules. This dissertation describes the design and implementation of PoliSeer and a case study in which we have used PoliSeer to specify and enforce a policy on PoliSeer itself.

PoCo is a language for specifying composable software-security policies. PoCo users specify software-security policies in terms of abstract input-output event sequences. The policy outputs are expressive, capable of describing all desired, irrelevant, and prohibited events at once. These descriptive outputs compose well: operations for combining them satisfy a large number of algebraic properties, which allows policy hierarchies to be designed more simply and naturally. We demonstrate PoCo's capability via a case study in which a sophisticated policy is implemented in PoCo.

CHAPTER 1

INTRODUCTION

General-purpose computers are only useful insofar as they execute software. Software is only useful insofar as it serves the needs of its user. When users execute programs, they do so with certain expectations of their behavior. For example, it is expected that an email client only delivers a message to its addressees and not to unspecified parties. Failure to meet the user's requirements may be malicious attacks or unintentional bugs. In any case, software better serves the user when s/he can explicitly codify his or her requirements and force a program to obey them. This codification constitutes a *software-security policy*—a restriction on program behavior. A *runtime monitor* is a mechanism that intercepts the instructions that the untrusted application tries to execute, checking the policy each time to determine the appropriate course of action. A monitor *enforces* a policy on an application if it ensures that every instruction sequence the program could attempt is made to conform to the policy.

Although complex software-security policies are difficult to specify, understand, and update, they arise often in practice. For example, a system administrator or end user may wish to enforce a complex collection of constraints (i.e., a policy) on an untrusted application to limit that application's access to resources such as files, memory, and peripheral devices, and to obligate the untrusted application to audit security-relevant operations and employ appropriate cryptographic protocols on network communications. In general, software-security policies tend to become more and more complex over time, due to the emergence of new attacks, users' demands for relaxations to overly tight policy constraints, and the

development of new application areas, like medical databases, which require domain-specific security and privacy considerations [3].

1.1 Related Work

The trend of increasing complexity in software-security policies mirrors the trend of increasing complexity in general software applications; however, many tools and techniques exist to help software engineers specify, analyze, and modify complex software applications. One of the most common techniques is modularization; engineers can modularize software into independent, reusable components (e.g., packages, classes, functions, aspects, etc.) that can be parameterized by, and can communicate with, other components through well-defined interfaces. Decomposing complex software into simpler modules saves engineers from having to manage software as a single, indecomposable code block. Integrated development environments (IDEs) for software engineering typically provide good support for navigating software modules [10, 34, 35, 38].

In contrast, recent efforts at creating tools for helping policy engineers specify arbitrary runtime policies have only permitted the management of indecomposable policies [22, 20, 11, 9, 12, 13, 33, 21, 36, 17]. These tools enable engineers to specify an arbitrary runtime policy as an isolated and centralized policy module; however, the tools do not enable engineers to decompose that centralized policy into simpler subpolicy modules, which could be specified, analyzed, reused, tested, and modified in isolation.

Other related efforts do allow users to specify, visualize, analyze, and/or compose policies, but only in particular domains. For example, the Policy Visualization Analysis tool provides a GUI for managing existing SE Linux policies [41]; the Expandable Grid manages access-control policies [31] (which are a proper subset of runtime-enforceable policies [24]); the Policy Mapper manages location-based access-control policies [6]; front ends for SPARCLE and PERMIS manage natural-language access-control and privacy policies [7, 18]; and Fang and Firmato manage firewall policies [28, 1].

The Polymer project has attempted to address the lack of tools for managing compositions of arbitrary runtime policies [3, 4]. Polymer is a language and tool for specifying and enforcing runtime policies on untrusted Java-bytecode applications. Polymer policies exhibit *universal composability* (every policy can be composed with other policies). Polymer achieves universal composability by (1) making all policies *first-class objects* (i.e., objects that are treated like all other values, which can be passed as arguments to and returned as results from methods) and (2) requiring all policy objects to implement a standard interface. Hence, a Polymer policy P can be parameterized by another policy P'; when P has to decide whether and how to allow a security-relevant application event A to occur, P may query P' for a response to A and use that response to generate its own response. For example, a `Conjunction` policy might take two policy arguments P_1 and P_2 in its constructor; the overall policy can enforce the conjunction of P_1 and P_2 by always responding to security-relevant events with the most restrictive of the responses of P_1 and P_2. In this case we call `Conjunction` a *superpolicy* and P_1 and P_2 its *subpolicies*. As another example, an `Audit` superpolicy may be parameterized by a policy P and a string S; then `Audit` can blindly enforce P while logging all of P's responses to security-relevant events in a file named S. Using such techniques of parameterizing policies with other policies, engineers can use Polymer to build complex runtime policies as compositions of simpler subpolicy modules (Figure 2.6 provides a high-level view of a complex Polymer policy that specifies runtime constraints on email clients and is built by composing simple subpolicy modules).

Although Polymer enables arbitrarily complex runtime policies to be specified more conveniently as compositions of subpolicies, Polymer achieves this convenience by requiring each individual policy module to be specified surprisingly inconveniently. The inconvenience stems from Polymer's requiring users to adhere to a complex programming discipline designed to partition policy code into effectless (i.e., free of state updates and I/O operations) and effectful methods. Figure 1.1 (which is based on a policy downloaded from the Polymer project's website [5]) shows the convoluted way policy logic must be specified in Polymer

```
public class ConfirmAllHTTP extends Policy {
    private boolean userCancel = false, noAsk = false;
    public Response query(Action a) {
        aswitch(a) {
            case ⟨abs void NetworkOpen(String addr, int port)⟩:
                if (port==80 || port==443) {
                    if (noAsk) return new OK(this);
                    if (userCancel) return new ExceptionResponse(this);
                    return new InsertResponse(this,new Action(null,
                        "JOptionPane.showConfirmDialog(Component, Object,
                            String, int)",
                        new Object[ ]{null, "Allow HTTP to "+addr+"?",
                            "Warning",
                            new Integer(JOptionPane.YES_NO_OPTION)}));
                }
        }
        return new IrrelevantResponse(this);
    }
    public void accept(Response r) {
        if (r.isExn()) userCancel = false;
        if (r.isOK()) noAsk = false;
    }
    public void result(Response r, Object rslt) {
        if (r.isIns() && ((Integer)rslt).intValue()==JOptionPane.NO_OPTION)
            userCancel = true;
        else if (r.isIns() && ((Integer)rslt).intValue()==JOptionPane.YES_OPTION)
            noAsk = true;
    }
}
```

Figure 1.1. Convoluted but composable Polymer policy requiring user confirmation before making HTTP connections (taken from [5]).

in order to make policies safely composable. As this example illustrates, specifying policies in Polymer requires care and expertise.

In Polymer, policies respond to input *actions* (method invocations), which produce *results* (return values), by outputting *suggestions*, which represent the policy's willingness to proceed with the input action. Some suggestions are *IrrSug* (the policy considers the action irrelevant), *OKSug* (the action is relevant, but permitted), and *InsSug* (some other action should be executed first before reconsidering the input action). Suggestions are so-called because only one will be followed per input action; the other policies are effectively ignored. When a suggestion is followed, the monitor notifies its suggester(s) beforehand providing an opportunity to update policy state. When following a suggestion entails executing an action, the action's result is relayed to the suggester(s) for further bookkeeping. This convoluted control-flow model makes policies harder to reason about, particularly when InsSugs are involved. Polymer policies are only able to emit suggestions for actions, so they cannot enforce the same properties on results as they can on actions.

Polymer policies being limited to emitting a single suggestion for each input has some significant consequences:

1. Policies cannot express more complex intentions such as a preference for some events combined with explicit opposition to some others without being explicitly asked about each.

2. Combinators are generally incapable of combining the intent of their subpolicies because some suggestions are parameterized over actions, results, or exceptions and there is no given way to combine these parameters.

3. Combinators typically output one of their subpolicies' suggestions and discard all others, sometimes at the expense of commutativity (e.g., `Conjunction` on two InSugs [4]). Therefore, combinator-policy organizations are often more sensitive to ordering than the policy author would like, and distinguishing such subtleties is an additional burden.

Policy-specification languages exist in which policy intent is more readily combined, but the policies are limited to access-control. *PBel* policies, for example, employ Belnap logic in which opposing Boolean values can be combined into either $\top$ ("conflict") or $\bot$ ("bottom"), representing the disjunction or conjunction of true and false, respectively [8]. *Ismene* policies allow entities to specify requirements and restrictions on their interactions with others over a network (e.g., requiring the use of a certain cryptographic algorithm or preventing those without sufficient authority) [29]. These requirements and restrictions of each entity are combined to produce a global policy for governing the overall session in a manner that respects them via a "reconciliation" process. *Tempura* is a simulation tool for examining policies written as Interval Temporal Logic formulae which can express safety properties whose permission decisions can depend on a discrete clock time [39]. ITL expresses grants and denials as signed "authorizations" which can be combined giving preference to one or the other or by reporting an error.

The problem of formal verification is closely related to policy enforcement in that both aim to ensure the conformity of a system to a specification. Formal verification approaches often employ some form of temporal logic to express restrictions on how a system may evolve from state to state [30]. Linear Temporal Logic (LTL), for instance, is a system for expressing formulae over propositions (Boolean variables) whose values may change on each discrete "step" of time. A state is a snapshot of the proposition values. An LTL formula is either satisfied or not by a given sequence of states, but has no power to modify that sequence.

Aspect-oriented languages, such as AspectJ, allow program code to be modularized according to logical categories (typically called "concerns" of the program) of which a security policy is one. Although most programming languages have some means of grouping logically related code (e.g., packages, classes, and methods), *cross-cutting concerns* are facets of program behavior that span across several such boundaries by nature. For example, the task of updating a log when certain behaviors occur is a famous cross-cutting concern. Suppose that we want to log every file access, we would normally have to insert and maintain logging code everywhere that a file access occurs, which is inconvenient. In AspectJ, an *aspect* implements a concern by defining a *pointcut* (pattern that matches relevant code) and *advice* (code specific to the concern) to be executed at *join points* (parts of the program matching the pointcut). However, AspectJ aspects are incomposable in that when multiple aspects offer advice at the same join point, execution of all advice proceeds according to universal precedence rules rather than by the programmer's discretion.

Mandatory Results Automata (MRAs) are expressive security mechanisms that monitor both the application's actions and the results of those actions [25]. MRAs are a more realistic model for runtime monitors than earlier security automata, e.g., Edit Automata [23], because they respect that programs generally receive a return value for each computation before moving on to the next one. The result-modifying capacity of MRAs further widens the gap between transformative monitors and those suitable only for access control. However, MRAs exist solely as a formal model.

1.2 Contributions

Policy specification, being a special problem domain, is well served by custom-tailored tools that address its concerns conveniently. This dissertation presents two such tools:

1. PoliSeer, a GUI-based utility designed to make it simpler and more convenient for non-experts to specify, visualize, modify, and enforce complex policies.

2. PoCo (Policy Composition), a language for specifying expressive runtime software-security policies in a modular fashion where individual policies compose well, combining their intentions comprehensively to avoid arbitrary choices and subtle policy-organization concerns.

As far as we are aware, PoliSeer provides the first GUI for composing arbitrary runtime security policies. PoCo constitutes a significant improvement in policy specification, building on lessons learned through experience with existing solutions.

PoliSeer users import universally composable policies from a policy library, compose them in meaningful ways by declaring arguments for all policy parameters, and generate code for the composed policy. We believe this process is straightforward enough that system administrators and even advanced end users can use PoliSeer to specify and enforce application-level policies; users simply customize (i.e., specify arguments for) expert-authored policies. Hence, the primary requirement for using PoliSeer is an ability to read and understand documentation for expert-authored policies.

Beyond specifying complex policies as compositions of simpler subpolicy modules, policy engineers can use PoliSeer to visualize complex policies holistically (as shown in Figure 2.6). Such high-level policy visualizations may improve the engineers' understanding of large and complex policies and help engineers locate and isolate problematic policy modules.

Our implementation of PoliSeer uses Polymer as the underlying language of universally composable policies; in other words, Polymer is the language in which our PoliSeer implementation imports and exports policies. However, we have partitioned the implementation

into Polymer-specific and non-Polymer-specific modules to make PoliSeer readily portable to other policy-specification languages with first-class and parameterized policies.

PoliSeer is a graphical, Turing-incomplete policy-specification tool, so it lacks the expressiveness of Turing-complete policy-specification languages like Polymer [3, 4], Naccio [13], and PSLang [12]. Instead, PoliSeer users must rely on expert policy engineers to make useful universally composable policies available. This sort of trade-off between expressiveness and usability is common with security-management tools. We believe that PoliSeer strikes a good balance between expressiveness and usability because, like standard IDEs, engineers may make use of PoliSeer's convenience when it does not impede expressiveness, but when greater expressiveness is required, PoliSeer users can always specify Polymer policies on their own and then import them directly into PoliSeer.

PoCo users specify software-security policies by declaring the sequential patterns of monitor inputs and outputs that are matched by all satisfactory program executions. These sequences support repetition and switching operations to incorporate loops and branches in the policy logic. Policies are free of externally observable effects by design, so policy authors don't need to rely on discipline to properly manage effectful computations (in contrast with Polymer's reliance on programmer discipline to manage effects).

PoCo policies specify input patterns as regular expressions that are matched against string representations of events input into the monitor. The output patterns are positively or negatively signed regular expressions (or combinations thereof) that describe the desirable and unacceptable output events, respectively, at that stage of the execution. Signed regular expressions allow policies to describe the whole set of favorable and unfavorable outputs at once as well as those they are indifferent toward (the strings in neither category). This feature is particularly useful in that negatively-signed expressions with infinite languages can be used to object to broad categories of events all at once. Signed regular expressions can be operated on similarly to their unsigned counterparts.

Regular languages are appreciated for satisfying a number of closure properties, which are useful for principled policy combination. For example, conjunction, disjunction, and in-

version are common policy-combination strategies, and regular languages are closed under intersection, union, and complementation. Properties like the commutativity and associativity of intersection and union operations free policy authors from considerations about which policy is on which side of a conjunction, for example. Removing opportunities for subtle corner-case behaviors improves policy correctness and reduces specification effort.

1.3 Summary of Relationships Between PoliSeer, PoCo, and Previous Work

Whereas existing tools for policy specification, visualization, analysis, and/or composition such as Firmato and Expandable Grids are for particular domains such as firewall and access-control policies, PoliSeer is for arbitrary runtime software-security policies. PoCo policies have a more straightforward control flow than those in Polymer because PoCo policies are aware of all monitor inputs and respond to both actions and results in the same manner. PoCo policies are also able to express their intent completely in a way that can be directly combined with others', unlike Polymer's suggestions and AspectJ's advice.

Although PBel policies, Ismene policies, and Interval Temporal Logic formulae compose well, they are limited to specifying access-control policies, whereas PoCo policies compose well and are more expressive. PoCo is influenced by the LTL's sense of sequentially ordered requirements and declarative nature, but its policies go beyond expressing what must hold to include how to affect the relationship between input and output events to make it so. PoCo is designed such that specifiable polices describe MRA executions and are MRA-enforceable.

1.4 Roadmap

We proceed as follows. Chapter 2 describes the design of the PoliSeer GUI. Chapter 3 explains and evaluates our implementation of PoliSeer as a Java application that inputs and outputs Polymer policies and reports our experiences implementing a case-study policy in PoliSeer. Chapter 4 details the principles, goals, and requirements of the PoCo policy-specification language. Chapter 5 introduces PoCo's linguistic constructs along with illustrative examples. Finally, Chapter 6 concludes.

CHAPTER 2

THE POLISEER INTERFACE

PoliSeer aims to provide a straightforward graphical interface for conveniently and flexibly managing complex policies.

2.1 The Main Window

The main PoliSeer window consists of two panels, as shown in Figure 2.1. The left panel is the *policy-selector panel*; the right panel is the *policy-tree panel*.

1. The policy-selector panel allows the user to navigate the machine's file system to find existing composable policies. Our implementation begins by populating the policy-selector panel with all subdirectories and `.poly` files (i.e., Polymer policy files) in the user's home directory. The policy-selector panel makes use of a standard interface for navigating the file system: clickable areas expand and contract subdirectories. When a user expands a subdirectory, PoliSeer searches for, parses, and displays all policy files in the newly visible directory. PoliSeer parses the policy files so that it can display the types of parameters each policy expects (in its constructor) next to that policy's name in the policy-selector panel, as shown in Figure 2.1 (when multiple constructors exist for the same policy, users select the desired constructor from a drop-down list before inserting the policy into a policy tree). Because computer users are accustomed to this sort of expand-and-contract navigation interface (e.g., the Windows file explorer and many application programs employ the same interface), navigating policy libraries is straightforward.

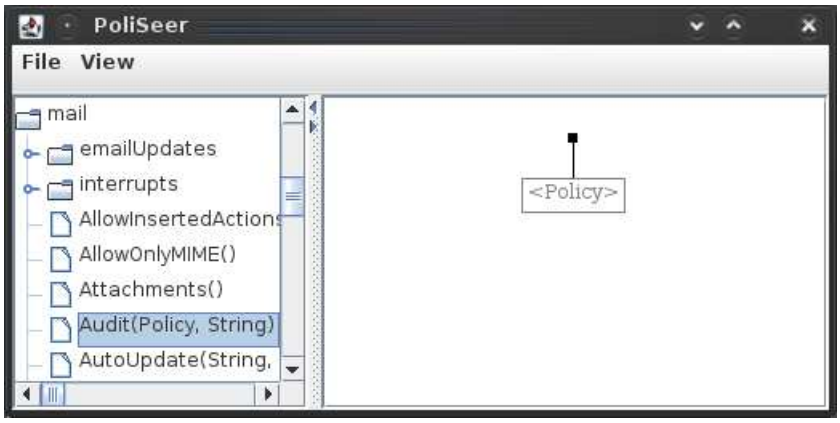

Figure 2.1. Main PoliSeer window divided into policy-selector and policy-tree panels. The policy-tree panel is displaying the default, empty policy.

2. The policy-tree panel contains a graphical representation of the policy currently being created, visualized, or modified. When PoliSeer begins executing, it displays the empty policy as shown in Figure 2.1. The empty policy consists of a single grayed-out node containing the text `<Policy>`, which indicates that PoliSeer expects that node to be filled in with a policy. In general, grayed-out nodes in a PoliSeer policy indicate incompletions in the policy; the text in a grayed-out node indicates the type of data that must be inserted into that node. In this way, PoliSeer communicates to the user whether, and in what ways, policies are incomplete. For example, Figure 2.2 shows a policy-tree panel for an incomplete, one-node `Audit` policy parameterized

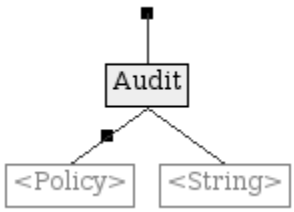

Figure 2.2. Policy-tree panel showing a root `Audit` policy parameterized by another `Policy` and a `String`, though no children have yet been specified.

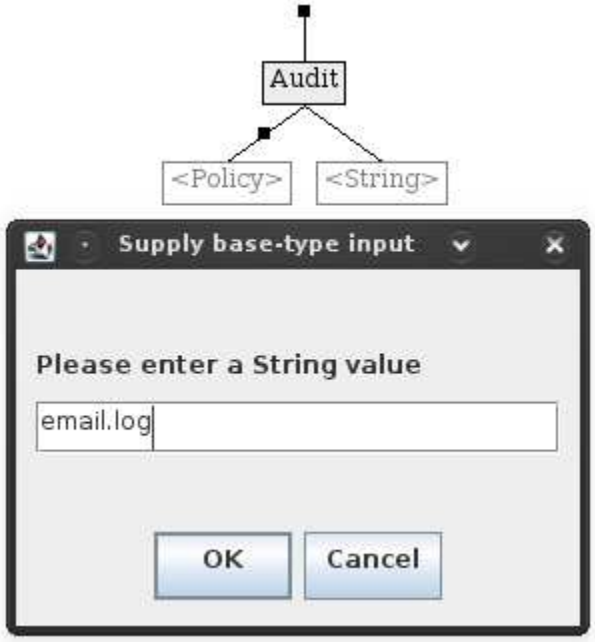

Figure 2.3. Policy-tree panel as the user enters a `String` argument for the `Audit` policy.

by another `Policy` and a `String`; the policy is incomplete until the user specifies one `Policy` and one `String` argument for `Audit`.

The only windows incorporated into PoliSeer besides the split main window are modal popup windows (for routine operations like selecting a location to load a policy from or save a policy to) and a window for viewing policy source code (described in Section 2.3).

2.2 Creating Policies

PoliSeer's basic interface for creating policies is simple. Users may select a policy in the policy-selector panel by left-clicking on the policy name. Having clicked on a policy P in the policy-selector panel, the user may left-click on any *landing area* L in the policy-tree panel to insert P into L. Valid landing areas are grayed-out `<Policy>` nodes and *branch-insertion points* (BIPs) in the policy-tree panel. PoliSeer automatically displays BIPs as small black squares in the policy-tree panel on every branch into which a user could possibly insert a policy.

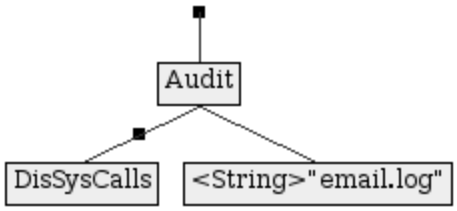

Figure 2.4. Policy-tree panel showing an `Audit` policy with a subpolicy and string argument. This complete policy disallows system calls (i.e., `java.lang.Runtime.exec` methods) at runtime while logging all policy decisions to a file named `email.log`.

For example, Figure 2.1 shows PoliSeer as it begins, with an empty policy-tree panel. A user may add the `Audit` policy as the root of the policy tree by clicking on the `Audit` policy in the policy-selector panel and then clicking on the grayed-out `Policy` node in the policy-tree panel. The policy tree in Figure 2.2 results from this addition; `Audit` has been added as the root node of the policy, but two new grayed-out nodes have appeared because PoliSeer has parsed the `Audit` policy and determined that it is parameterized by another `Policy` and a `String`. The user may then insert a `String` as the right child of the `Audit` policy by clicking on the grayed-out `String` node and entering the string in a pop-up window, as shown in Figure 2.3. Users may enter other types of arguments to policies, such as `int`s, `float`s, `boolean`s, and `char`s, similarly to `String`s, but PoliSeer will first confirm that the user's entry can be parsed as a value of the correct type. Currently, PoliSeer can only import and manipulate policies with a primitive-type `String` and `Policy` parameter, but all the policies provided in the standard Polymer distribution [5] satisfy this constraint.

Continuing with this example, Figure 2.4 shows a complete policy tree that results from inserting a (childless) policy and a string into the grayed-out nodes of Figure 2.2. Two BIPs exist in Figure 2.4; a user may insert a policy node into this policy above the `Audit` root or above the `DisSysCalls` child of `Audit`. To insert a `Conjunction` policy between the `Audit` and `DisSysCalls` nodes, the user simply clicks on the `Conjunction` policy in the policy-selector panel and then clicks on the BIP between the `Audit` and `DisSysCalls` policies in Figure 2.4; the result is shown in Figure 2.5.

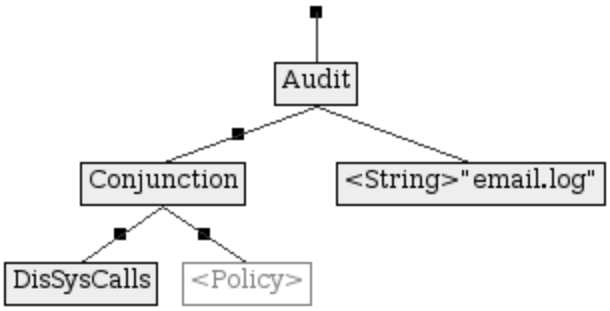

Figure 2.5. The same policy-tree panel shown in Figure 2.4, except that the user has now inserted a `Conjunction` policy between the `DisSysCalls` and `Audit` policies.

By building policy trees, PoliSeer users may specify complex policies. All the superpolicies in policy trees can be thought of as *metapolicies*; superpolicies like `Conjunction` specify how to combine any possible combination of constraints imposed by subpolicies, even when those constraints conflict. For example, a PoliSeer user may specify a policy as being the conjunction of two subpolicies P_1 and P_2, where P_2 is defined to always respond to security-relevant actions in the opposite way as P_1 (e.g., when P_1 OKs an action, P_2 halts the target, and when P_1 does anything besides OK an action, P_2 OKs the action; this is the `Not` superpolicy described in Section 4.1). Such a composition of conflicting policies is perfectly legal, and it is up to the semantics of the superpolicies to determine what happens when policies specify conflicting constraints. In this case, the `Conjunction` superpolicy obeys the strictest of its subpolicy constraints on each security- relevant action, so the overall (`Conjunction`) policy will always obey P_1's constraints as long as P_1 does not OK an action, but when P_1 OKs an action, the overall `Conjunction` will have to obey P_2 and halt the application. The ability to use metapolicies (i.e., superpolicies) to resolve conflicts between composed subpolicies is one of the key benefits of universally composable policies in systems like Polymer.

Having created a (complete or incomplete) PoliSeer policy, a user may save it to a `.psr` file (which is simply a serialization of the policy tree) with the `File -> Save Tree` menu option and may generate ready-to-enforce Polymer code for the policy in a `.poly` file with

the `File -> Generate Policy Code` option. Conversely, users may resume creating, visualizing, or modifying a saved `.psr` policy with the `File -> Load Tree` option. When exporting an incomplete PoliSeer policy to a `.poly` file, PoliSeer automatically parameterizes the exported policy by all missing policy components (e.g., if the policy is missing one child of a `Conjunction` superpolicy, then the exported policy's constructor will accept a `Policy` argument to fill in for that missing child).

2.3 Visualizing Policies

As Figures 2.6–2.8 demonstrate, PoliSeer's policy-tree panel can provide a useful high-level visualization of complex security policies as compositions of simpler subpolicy modules. If PoliSeer's visualization of a policy is too high level, users can always choose the `View -> Policy Source` menu option to obtain the source-code-level details of the most recently selected policy. Examining a policy's source-code documentation can be helpful for PoliSeer users when figuring out which arguments to specify for that policy. For example, a PoliSeer user may see and be intrigued by the `Audit` policy, view the documentation in the `Audit` source code, understand what the policy does and which arguments it expects, include `Audit` in the policy tree, and supply appropriate child-node arguments based on an understanding of `Audit`'s semantics.

As another aid for visualizing policy compositions, PoliSeer provides a toggleable menu option `View -> Show Non-Policy Nodes`. This option removes (or restores) non-policy nodes in the policy-tree panel. Removing non-policy nodes from a policy tree may simplify the user's view of a policy, as Figures 2.6 and 2.7 demonstrate. Non-policy nodes often clutter a policy tree without providing much insight into the policy's organization. For example, non-policy nodes may specify port-number, IP-address, or filename arguments to policies, which may be irrelevant for understanding the overall policy structure.

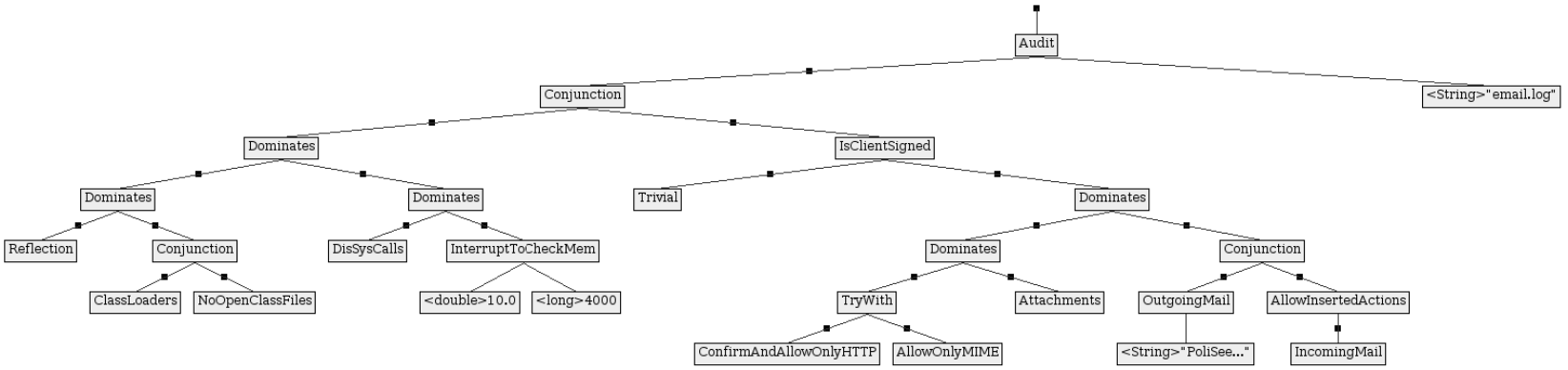

Figure 2.6. Full tree for an example email policy (taken from [3, 4]).

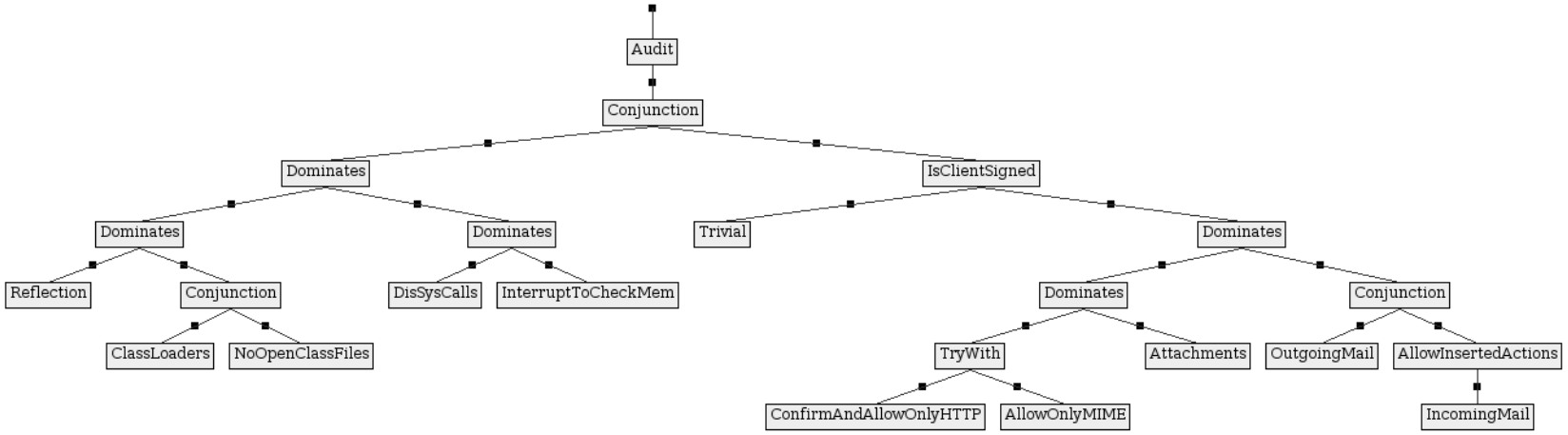

Figure 2.7. The same policy tree shown in Figure 2.6 but simplified by hiding non-policy nodes.

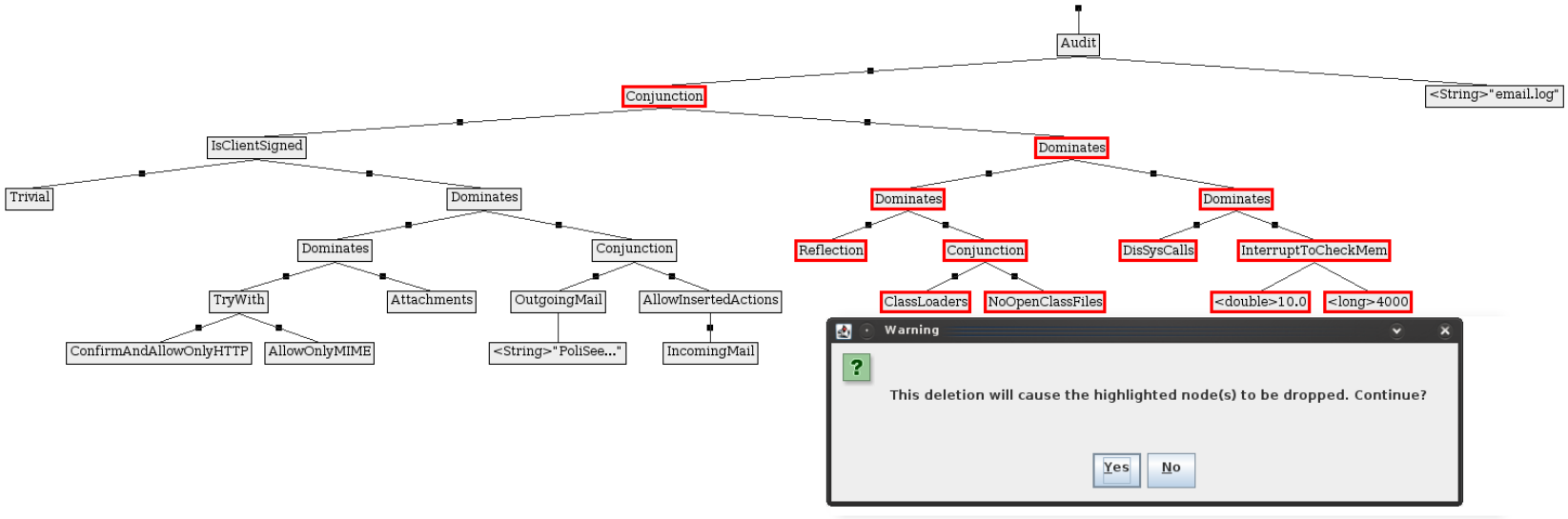

Figure 2.8. Warning displayed before deleting a node with multiple policy children. PoliSeer has highlighted the descendants that will be discarded along with the node being deleted.

2.4 Modifying Policies

Although we have found PoliSeer's interface for creating and visualizing policies convenient and straightforward, we have found policy-modification operations more nuanced and challenging to enable and implement.

PoliSeer users may modify policy trees in three ways:

1. Users may swap two existing sibling nodes (and their subtrees) by dragging and dropping one sibling node on another. Swapping subpolicies can be useful when dealing with superpolicies that make semantic distinctions between the order of their children (e.g., Polymer's `Dominates` superpolicy gives priority to its left child [3, 4]). Nodes must have the same type to be swapped (e.g., a `String` cannot be swapped with a `Policy`), and PoliSeer currently does not support non-sibling node swapping due to policy-tree circularities that arise when swapping a node with one of its ancestors.

2. Users may replace a policy node P in the policy tree with a policy P' selected from the policy-selector panel by left-clicking P' in the policy-selector panel and then left-clicking on P in the policy tree. PoliSeer only allows P' to replace P when the parameter types of P and P' are *well aligned*. Technically, this means that PoliSeer must be able to assign each of P's nonempty children to be children of P' without introducing any type conflicts. If P and P' are well aligned, PoliSeer performs the replacement by making the P node be a P' node and then traversing P's children from left to right and reassigning each nonempty child of P to the leftmost child of P' with the same type. In this way, PoliSeer attempts to allow policy replacement in all cases in which it could possibly make sense.

3. Users may delete a policy-tree node by right-clicking on it. Before deleting any nonempty policy-tree node, PoliSeer confirms the deletion with a popup dialog box. When deleting a policy node N, the leftmost policy-child of N takes the place of N in the policy tree, and PoliSeer discards all other children of N. Because users may

not expect this deletion semantics, PoliSeer highlights all the about-to-be-discarded nodes and displays a confirmation window before actually discarding the highlighted nodes. Figure 2.8 illustrates this interface.

When combined with the ability to insert policy nodes into any landing area in a policy tree, these three operations provide users a complete palette of basic policy-specification, -visualization, and -modification tools.

CHAPTER 3

IMPLEMENTATION OF POLISEER

We have implemented PoliSeer as an open-source Java application, available online at `http://www.cse.usf.edu/~ligatti/projects/poliseer/`. The implementation is 3351 lines of code in 12 source-code files.

3.1 Architectural Overview

Our PoliSeer implementation consists of three high-level modules:

1. The front end (1828 lines of code). This module reads and parses Polymer files for input into PoliSeer. When a user opens a directory in the policy-selector panel, PoliSeer parses the Polymer files in that directory to determine how they are parameterized, that is, which types of arguments the policies' constructors expect to receive. PoliSeer uses this type information to ensure that all policies receive arguments of the proper types and to prepare grayed-out children in the policy tree, as discussed in Section 2.2. Our front end parses Polymer files with a parser generated by JavaCC, a top-down parser generator [19].

2. The PoliSeer GUI (1451 lines of code). This module contains all the code to implement PoliSeer's graphical user interface, as described in Chapter 2.

3. The back end (72 lines of code). This module generates Polymer code for the policy tree being visualized. Users can input the code that this module generates directly into the Polymer system, which will then enforce the specified policy on untrusted Java-bytecode applications.

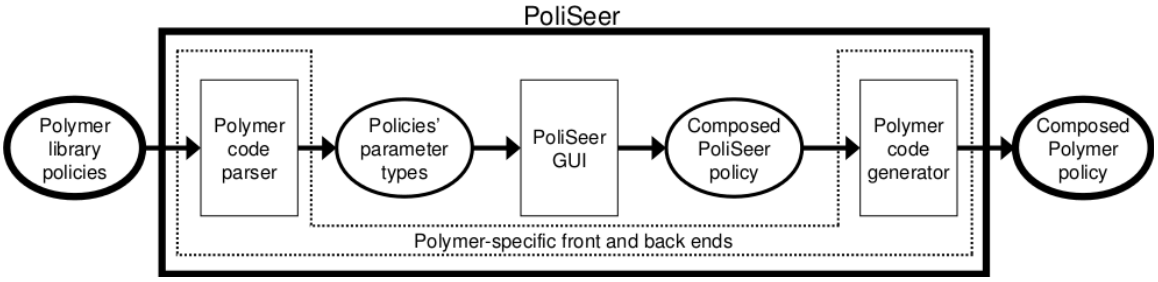

Figure 3.1. Architectural overview of PoliSeer.

Figure 3.1 summarizes PoliSeer's implementation architecture. The Polymer-dependent front and back ends are distinctly separated from the Polymer-independent GUI, so developers can change PoliSeer's underlying policy-specification language from Polymer to a language L by writing and plugging in new front and back ends for L.

3.2 Performance

We have tested our implementation's performance during basic operations such as parsing Polymer files and inserting nodes into policy trees. The tests were performed on a Sony Vaio laptop with Intel Core2 Duo 1.73GHz CPUs and 1GB of RAM, running Kubuntu 8.10. For all tests, we report running times obtained by averaging real execution times over ten executions.

Our first test measured the time taken for PoliSeer to start up, build the GUI, and exit at the end of PoliSeer's `main` method. This time included the virtual-machine start-up time and was just 830ms on average.

Next, we measured the time taken for PoliSeer to parse the Polymer files in the policy library to determine the types of parameters in the policies' constructors. With an average Polymer-policy-file size of 84.5 lines of code, PoliSeer parsed the Polymer files in only 3.1ms on average.

Our third test measured the amount of time PoliSeer took to render a policy tree during insertion of nodes into the tree. This rendering time dominates the amount of time it takes for PoliSeer to insert new nodes into policy trees; node-insertion time is $O(\lg n)$ for finding

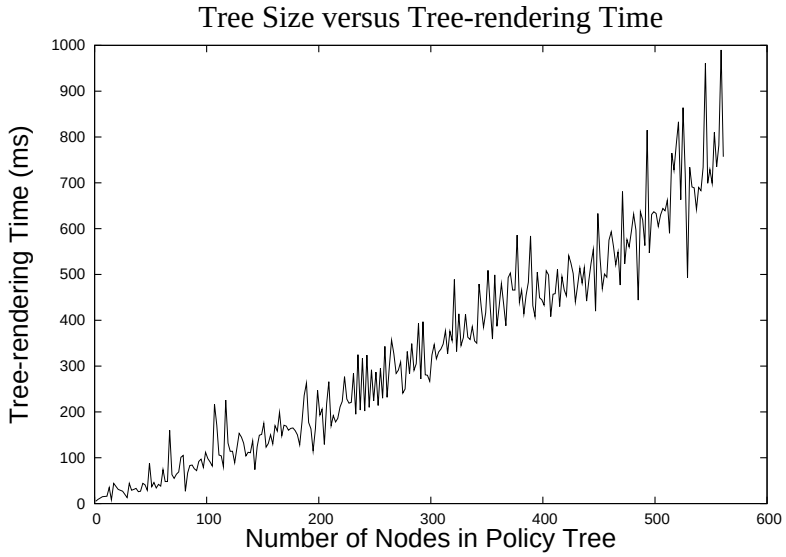

Figure 3.2. PoliSeer performance rendering policy trees during node insertion.

where to modify the tree data structure, O(1) for modifying the tree at that point, and O(n) for rendering the new tree with the inserted node (where n is the number of nodes in the policy tree). Figure 3.2 confirms the linear growth of policy-tree rendering. All tree-rendering-intensive operations in PoliSeer (i.e., node insertion, swapping, replacement, and deletion) exhibit a performance similar to that shown in Figure 3.2. Although tree-rendering times never exceeded one second, even for trees with hundreds of nodes, a good optimization to consider in the future would be to only re-render the modified portions of trees during policy-tree manipulations.

Finally, we measured the time taken for PoliSeer to generate Polymer code files from policy trees. We implement code generation by (recursively) preorder-traversing the policy tree, while concatenating strings to construct every policy-tree node. As Figure 3.3 illustrates, PoliSeer's code-generation time remains low (less than a second) even for policies with many hundred nodes.

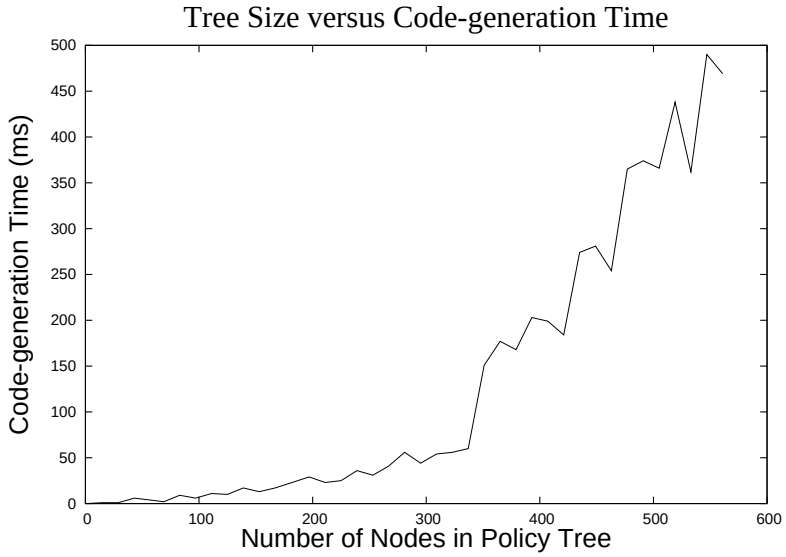

Figure 3.3. PoliSeer performance generating Polymer code.

In summary, all the basic PoliSeer operations have tolerable performance, and performance delays do not even become noticeable until the user manipulates policy trees containing several hundred nodes.

3.3 Case Study: Policy Overview

We have designed a complex case-study PoliSeer policy that restricts the runtime behavior of PoliSeer itself; that is, we have created a PoliSeer-controlling policy in PoliSeer. Moreover, we have successfully executed PoliSeer in the Polymer system while enforcing this PoliSeer-created policy.

Figure 3.4 displays the policy tree for our case-study policy. This policy has a `Conjunction` as its root, so it constrains the untrusted application (PoliSeer in this case) by always responding to a security-relevant action with the most restrictive of its subpolicies' responses to the same action. In other words, the case-study policy always attempts to respect the restrictions of two high-level subpolicies.

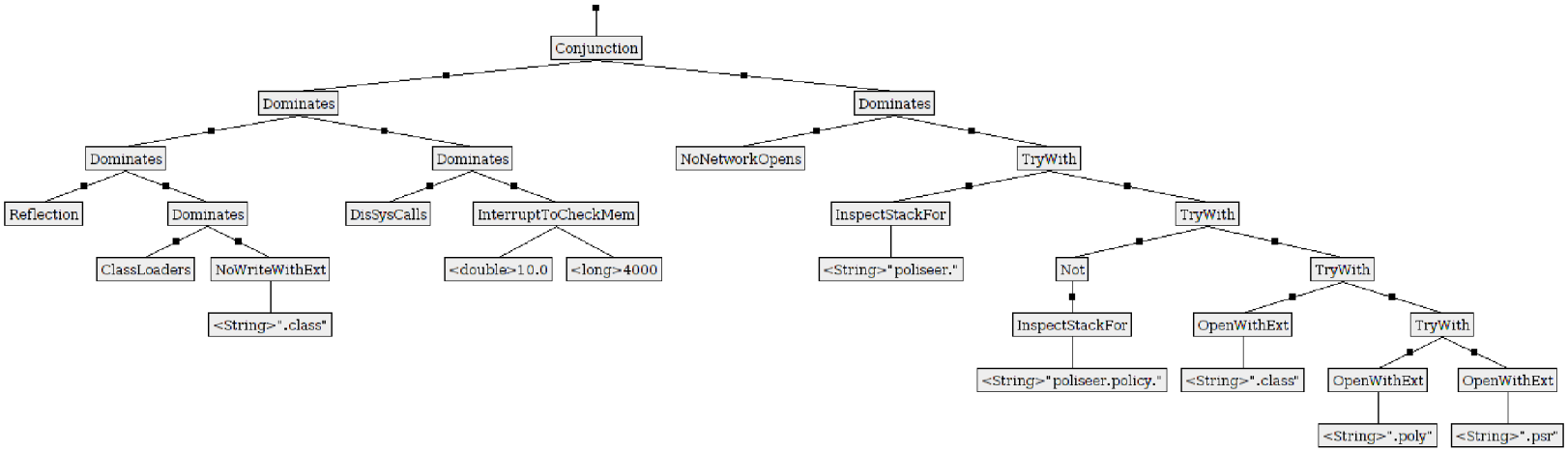

Figure 3.4. Case-study policy, specified in PoliSeer, that constrains the PoliSeer application itself.

We based the first of these two high-level subpolicies on policies described in earlier work [3, 4, 5]. This branch of the case-study policy enforces constraints that should be included in all Polymer policies to prevent the untrusted application program from using reflection, constructing class loaders, writing `.class` files, or invoking system-level functions (with `java.lang.Runtime.exec` methods). This branch also includes the `InterruptToCheckMem` policy, which notifies the user if the virtual machine's memory consumption exceeds a specified threshold. All of these subpolicies are conjoined by `Dominates` superpolicies, which act as short-circuit `Conjunction` policies in our case study (though their precise semantics is more subtle [3, 4]).

The second of the two high-level subpolicies in our case-study policy specifies constraints that we particularly wanted to enforce on PoliSeer; something would be wrong if PoliSeer violated any of these constraints. We call this branch of policies the PoliSeer-specific policy. Like the entire case-study policy, the PoliSeer-specific policy decomposes into two branches, joined with a `Dominates` superpolicy (which again acts as a short-circuit `Conjunction` here).

The first branch of the PoliSeer-specific policy is the `NoNetworkOpens` policy, which disallows the untrusted application from opening any network sockets.

The second branch of the PoliSeer-specific policy restricts the PoliSeer application (i.e., code in the `poliseer` package)—but not the case-study policy we are enforcing on PoliSeer (i.e., code in the `poliseer.policy` package)—from opening files with extensions other than `.psr`, `.poly`, or `.class`. Intuitively, although the case-study policy may open other types of files (perhaps because, e.g., we later extend the policy with auditing capabilities that necessitate opening log files), the PoliSeer application itself should have no effect on the file system except for reading and writing PoliSeer (`.psr`), Polymer (`.poly`), and Java-bytecode (`.class`) files. The application will actually not be able to *write* `.class` files (because the `NoWriteWithExt` policy already disallows it), but the case-study policy does allow the application to *read* `.class` files (because Java compilers often optimize benign operations like initializing nested classes by having the outer class's initializer read the bytecode of the inner class).

This second branch of the PoliSeer-specific policy contains four subpolicies:

1. `TryWith` combines two subpolicies by first obtaining its left child's response to the security-relevant action A that the untrusted application is attempting to execute. If the left child allows A to execute unconditionally then so does the `TryWith` superpolicy; otherwise, `TryWith` responds to A with whatever response its right child returns for A.

2. `InspectStackFor` takes a `String` argument S and inspects the runtime call stack for a method called from package S. More specifically, the policy traverses the call stack from the most recent to the oldest method invocation and disallows the security-relevant action the untrusted application is about to execute if and only if the traversal reaches a call from S before reaching a `doPrivileged` call.

3. `Not` inverts the response of its subpolicy. For example, if the subpolicy responds to a security-relevant action by halting, the `Not` superpolicy will respond by allowing the action.

4. `OpenWithExt` takes a `String` argument S and allows file-open actions to execute if and only if they access files with names that end with file-extension S.

The case-study policy enforces the desired file-open subpolicy by chaining together `TryWith` combinators. Every child of the `TryWith` policies allows one type of file-open action to execute; a file open is only disallowed when none of those children allow it. In turn, the `TryWith` children allow actions from outside the `poliseer` package, actions from within the `poliseer.policy` package (technically, from `Not` outside the `poliseer.policy` package), actions that open `.class` files, actions that open `.poly` files, and actions that open `.psr` files.

3.4 PoliSeer Performance in Polymer

We measured the performance of PoliSeer executing in the Polymer system while enforcing the case-study policy. The measurements provide some insight into the runtime overhead induced by enforcing Polymer policies.

In general, runtime-policy-enforcement overheads depend on the complexity of the policy being enforced and the number of times that policy must respond to security-relevant actions. One important consideration in this regard is that our case-study policy only reasons about (i.e., considers security relevant) application methods that open files, open network sockets, make system-level calls, implement reflection, or construct class loaders. Enforcing the case-study policy induces runtime overhead only when the untrusted application (PoliSeer) invokes one of these security-relevant methods. Therefore, operations like inserting nodes into policy trees, which do not execute security-relevant methods, run equally quickly regardless of whether the case-study policy is being enforced. On the other hand, application-level operations that do execute security-relevant methods experience overhead when the case-study policy is being enforced, and that overhead is proportional to the number of security-relevant methods executed.

The average time for PoliSeer to start up in the Polymer system was 3.7s, much higher than the 0.83s start-up time we measured when the case-study policy was not being enforced. This significant start-up overhead is common in Polymer due to the large number of files that get opened as the virtual machine loads classes during application startup (recall that our policy considers file openings security relevant).

Besides the start-up overhead, none of the overheads induced by enforcing the case-study policy on PoliSeer were noticeable. The average time to parse Polymer files increased by 0.8ms, and the average time to generate Polymer code increased by 1.3ms (regardless of the policy size), when enforcing the case-study policy. The case-study policy has to respond once to each Polymer-file parsing and code-generation operation because these operations each entail one security-relevant file-open action. Hance, enforcing the case-study policy added about 1ms to the execution time of every security-relevant action.

CHAPTER 4

POCO PRINCIPLES, GOALS, AND REQUIREMENTS

This chapter presents the philosophical and technical materials that form the foundation of the PoCo (Policy Composition) language. The language's design is guided by a number of precepts born out of experience with existing policy-specification languages—mainly Polymer [5, 3, 4, 2]. The ultimate goal is to ease the overall burden of specifying complex, general-purpose policies by simplifying how policies are combined and individually specified without sacrificing expressiveness.

4.1 Execution Model and Terminology

First, we model the execution of a potentially dangerous program and introduce the relevant vocabulary. The model and terminology presented here is not original, but is necessary background for understanding the remainder of this dissertation. The form of PoCo policies is largely shaped by how the monitored execution of software is modeled. PoCo is designed to express policies that are enforceable by Mandatory Results Automata (MRAs) [25].

The MRA model, depicted in Figure 4.1, involves the following components:

1. Actions—objects that represent computations (e.g., function calls).

2. Results—objects that represent the return values of actions (e.g., primitive values or data structures).

3. Events—a generic term pertaining to both actions and results.

4. The executing system—an evaluator that outputs results for input actions.

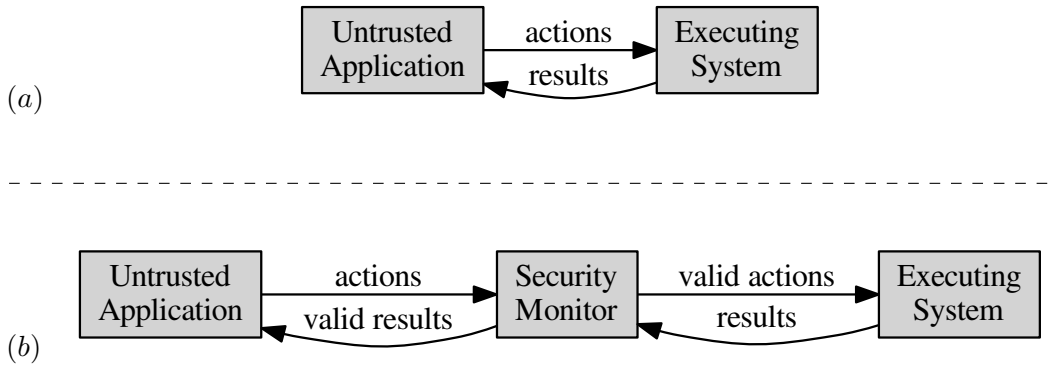

Figure 4.1. Application execution with and without a security monitor. An application executes normally (a) by sending actions to the executing system and receiving results from it directly. A security monitor regulates the actions sent to the executing system and results sent to the application (b) such that only valid executions occur. This is an interpretation of Fig. 1 from [25].

5. The untrusted application—a potentially dangerous program that executes by emitting actions and receiving their results.

6. The security monitor—a (potential) manipulator of the events transmitted between the application and executing system.

In the MRA model, the application sends actions to the executing system to evaluate, and the system replies to each action with a result. The monitor manages the exchange between the two, behaving as the system from the application's perspective and as the application from the system's perspective. When the monitor receives an action a from the application (the only possibility when execution begins), it can either output a to the system, output an alternative action $a' \neq a$ to the system, or output some result to the application. When the monitor receives a result r from the system, it can either output r to the application, output an alternative result $r' \neq r$ to the application, or output some action to the system. Results are called "mandatory" because the application and monitor can't output a new action until receiving a result for the last action it output.

The following notation and definitions, many of which are are introduced in [25] and are re-stated here, are used throughout the remainder of this dissertation:

1. Σ is the *alphabet*—a finite set of symbols.
2. $s_1 s_2 \cdots s_n$, where $s_i \in \Sigma, 1 \leq i \leq n$, is called a *string* over Σ.
3. ϵ is the zero-length *empty string*.
4. Σ^* denotes the set of all strings over Σ; Σ^+ is the same but omits ϵ.
5. Metavariable σ ranges over Σ^*.
6. $A \subset \Sigma^+$ is the nonempty, computable set of all actions.
7. Metavariable a ranges over A.
8. $R \subseteq (\Sigma^+ \setminus A)$ is the nonempty, computable set of all results of actions.
9. Metavariable r ranges over R.
10. $E = A \cup R$ is the set of all events.
11. Metavariable e ranges over E.
12. $\langle e, e' \rangle$ is an event pair called an *exchange*, where e is called the *input event* (i.e., an event input to the monitor) and e' is called the *output event* (i.e., an event output from the monitor).
13. ε denotes the set of all exchanges over E.
14. Metavariable x ranges over ε.
15. $x_1 x_2 \cdots x_n$, where $x_i \in \varepsilon, 1 \leq i \leq n$, is called a *finite execution* over E.
16. $x_1 x_2 \cdots$, where $x_i \in \varepsilon, 1 \leq i$, is called an *infinite execution* over E.

4.2 Design Principles

We believe that policies should not only be modular and freely composable with other policies, but that the composition of these policies should not introduce nuances and subtleties that make the resultant policy hierarchy difficult to reason about. Users should be able to export well-formed policy hierarchies in a tool like PoliSeer and import them as singular policies in yet more complex trees without worrying that organizational details in the former will have unexpected consequences in the latter. In short, well-formed policy trees should be totally encapsulated.

A violation of this principle can be illustrated with Polymer's `Disjunction` and `Conjunction` combinators, which generally output the least restrictive and most restrictive output of their two sub-policies, respectively. Recall that Polymer policies receive a `query` on each input action, and output a `Suggestion` that describes how execution should proceed. The `InsSug(`a`)` output is the first choice of both combinators because the original input will be re-considered after action a is executed. Suppose subpolicies p_1 and p_2 output `InsSug(`a`)` and `InsSug(`a'`)`, respectively, `Disjunction` will output the left-hand suggestion `InsSug(`a`)` only, even though `InsSug(`a'`)` is no more restrictive. This is because `InsSug` values are parameterized over a single action and only one `Suggestion` can be output by any single policy or combinator, so there is no way to actually combine the output of both subpolicies. `Conjunction` behaves similarly, but consolidates the outputs if the inserted actions are the same (i.e., $a = a'$)—the details of the consolidation are not essential to this example.

Now, consider the simple composition `Conjunction(Disjunction(`p_2, p_1`)`, p_1`)`. If $a \neq a'$ in the outputs of p_1 and p_2, `Disjunction` will output `InsSug(`a'`)` and `Conjunction` will choose `Disjunction`'s output, `InsSug(`a'`)`, because the two actions are not equal. If `Disjunction` were to output its right-hand suggestion instead, then `Conjunction` would be consolidating two `InsSug(`a`)` values. Therefore, this arbitrary local decision on the part of `Disjunction` impacts the behavior of higher-up policies (in this case, `Conjunction`),

which is difficult to reason about. This problem is amplified when complex compositions are collapsed into a single policy and these details are obscured.

Notice that the these basic Polymer combinators are not commutative when the sub-policies suggest inserting different actions. Failure to satisfy simple algebraic properties is a symptom of undesirable compositional behaviors. PoCo's design is informed by the following rules and guidelines, which we believe improve the policy-specification experience:

1. The basic combinators—conjunction, disjuction, and inversion—should exhibit desirable algebraic properties such as:
    (a) Conjunction and disjunction should be commutative, associative, and distribute over each other.
    (b) Inversion should distribute over conjunction and disjunction (De Morgan's Laws).
    (c) A doubly inverted policy should behave indistinguishably from the policy itself.
2. Policies should be:
    (a) free from externally observable effects as a consequence of the language design rather than by programming discipline.
    (b) aware of all inputs to the monitor, not relying on combinators to pass information down.
    (c) explicit and complete, not leaving unhandled corner cases.
    (d) decidable, permitting static analysis.
3. Policy outputs should:
    (a) wholly capture a policy's recommendations and prohibitions at once.
    (b) be directly combinable with others to create a new output that combines their meanings, distinct from each individual policy output.

(c) be able to prohibit infinitely many events at once.

(d) be enforceable by some decision procedure.

Seeking to satisfy the above conditions ultimately results in the PoCo language, which diverges significantly from Polymer. Improving the composability of policy outputs is a large step toward meeting these goals. The definition of the policy outputs dictates the combinators that are allowable on them, which impacts how base policies are specified and composed.

4.3 Signed Regular Languages As Policy Outputs

Regular languages come close to meeting the above criteria for policy outputs. They are closed under union, intersection, and complementation, so these natural combinators are supported. These operations also exhibit desirable algebraic properties (e.g., commutative disjunction). It can be decided whether some string is in a given regular language and, given a regular language, a string member can be generated (unless the language is empty, which is also decidable). Regular languages can also be infinite. However, regular languages partition the set of all strings into only two categories: members and non-members. Our purposes require three categories: positive members, non-members, and negative members, which signify the desired, irrelevant, and prohibited output events, respectively.

Definition 3.1 (*Signed Regular Languages*). Let Σ be some alphabet and $\mathfrak{R}$ be the set of regular languages over it, then $\mathfrak{R}^{\pm} = \{\langle A^+, A^-\rangle \in \mathfrak{R} \times \mathfrak{R}, A^+ \cap A^- = \emptyset\}$ is the set of signed regular languages over Σ. That is, a signed regular language is a pair of disjoint regular languages. Because ϵ is not an event, it is assumed that $\epsilon \notin (A^+ \cup A^-)$ throughout this dissertation.

The language in which all strings are neutral (i.e., non-members) $\langle \emptyset, \emptyset \rangle$ is called θ. θ is an important value in PoCo that occurs frequently in policies. As a policy output, it is similar to the Polymer's `IrrSug` ("irrelevant suggestion") value in that it expresses a lack of both favor and disapproval.

Definition 3.2 (*Signed Language Membership*). Let $A = \langle A^+, A^- \rangle$ be a signed regular language and $\sigma \in \Sigma^+$ be a string.

1. $\sigma \in^+ A \Leftrightarrow \sigma \in A^+$ (sim. $\sigma \notin^+ A \Leftrightarrow \sigma \notin A^+$)
2. $\sigma \in^- A \Leftrightarrow \sigma \in A^-$ (sim. $\sigma \notin^- A \Leftrightarrow \sigma \notin A^-$)
3. $\sigma \in^\pm A \Leftrightarrow ((\sigma \in^+ A) \vee (\sigma \in^- A))$ (sim. $\sigma \notin^\pm A \Leftrightarrow ((\sigma \notin^+ A) \wedge (\sigma \notin^- A))$)

Definition 3.3 (*Signed Deterministic Finite Automata (SDFAs)*). A signed DFA $\mathcal{M}^\pm$ over alphabet Σ is a pair of DFAs $\langle \mathcal{M}^+ = \langle \Sigma, Q, q_0 \in Q, \delta : Q \times \Sigma \to Q, F \subseteq Q \rangle, \mathcal{M}^- = \langle \Sigma, Q', q'_0 \in Q', \delta' : Q' \times \Sigma \to Q', F' \subseteq Q' \rangle \rangle$ such that $L(\mathcal{M}^+) \cap L(\mathcal{M}^-) = \emptyset$, where

1. Q and Q' are finite sets of states.
2. q_0 and q'_0 are initial states.
3. δ and δ' are transition functions.
4. F and F' are final states.

$\mathcal{M}^\pm$ is said to "positively accept" $\sigma \in \Sigma^+$ when $\mathcal{M}^+$ accepts σ, and "negatively accept" σ when $\mathcal{M}^-$ accepts σ; otherwise, σ is "rejected".

Note that the language of an SDFA—a pair of disjoint regular languages—is a signed regular language. Constructing an SDFA recognizing some signed regular language $\langle A^+, A^- \rangle$ is a matter of pairing a DFA recognizing A^+ with another DFA recognizing A^-. The requirement that the positive and negative components of a signed regular language be disjoint is a matter of consistency; no policy should both favor and prohibit any certain event at the same time. This constraint drives the definition of the basic signed regular operations.

Definition 3.4 (*Signed Regular Operations*). Let $A = \langle A^+, A^- \rangle$ and $B = \langle B^+, B^- \rangle$ be signed regular languages. The following operations are defined over them:

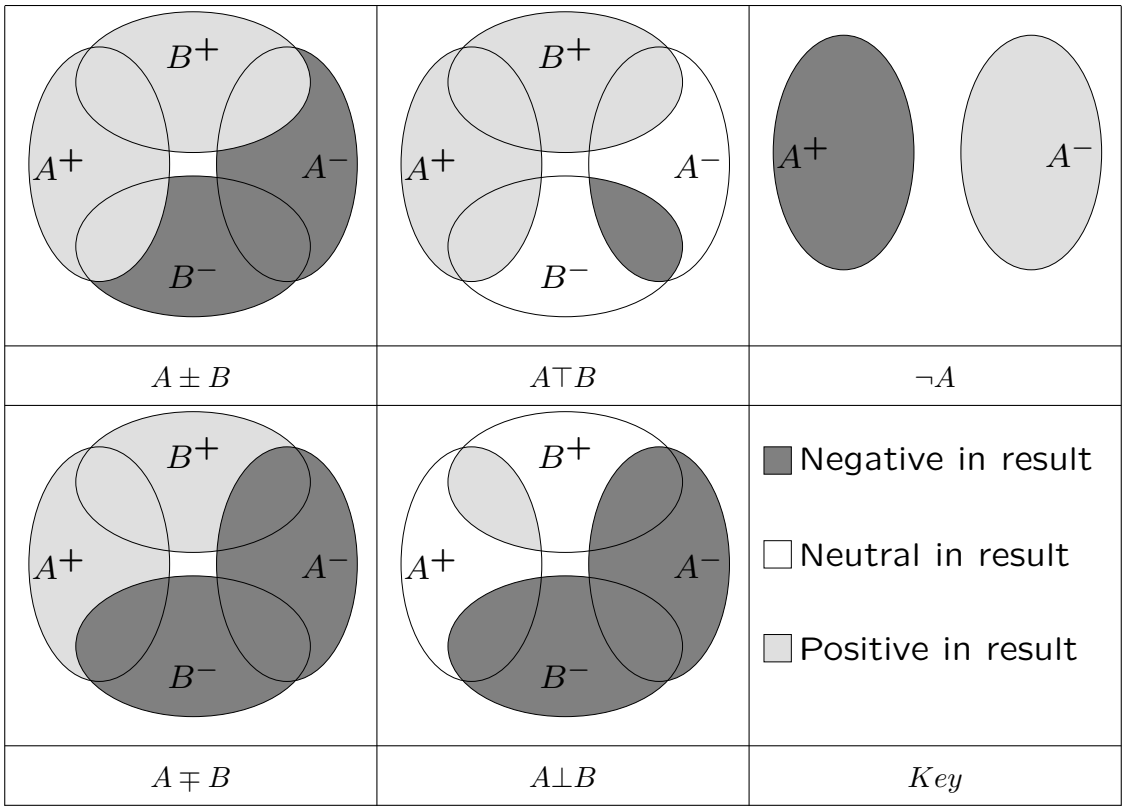

Figure 4.2. Shaded diagrams of the signed regular operations.

$$A \pm B \stackrel{def}{=} \langle A^+ \cup B^+, (A^- \cup B^-) \setminus (A^+ \cup B^+) \rangle \quad \text{(Optimistic Union)}$$

$$A \mp B \stackrel{def}{=} \langle (A^+ \cup B^+) \setminus (A^- \cup B^-), A^- \cup B^- \rangle \quad \text{(Pessimistic Union)}$$

$$A \top B \stackrel{def}{=} \langle A^+ \cup B^+, A^- \cap B^- \rangle \quad \text{(Disjunction)}$$

$$A \perp B \stackrel{def}{=} \langle A^+ \cap B^+, A^- \cup B^- \rangle \quad \text{(Conjunction)}$$

$$\neg A \stackrel{def}{=} \langle A^-, A^+ \rangle \quad \text{(Inversion)}$$

We can visualize the operations using a Venn-diagram-like technique in which the regions that are positive and negative members of the result are shaded differently, and those that are non-members of the result are unshaded. Figure 4.2 depicts the signed regular operations in this form. The gap in the center of each diagram corresponds to the non-members of both languages (i.e., $\overline{(A^+ \cup A^-)} \cap \overline{(B^+ \cup B^-)}$).

The signed regular operations capture common strategies for combining the intents of software-security policies. The $\pm$ and $\mp$ operations embody the desire to respect both the

preferences and prohibitions of two policies; they differ in terms of whether preference or prohibition is favored whenever there is some conflict between the two outputs. The $\perp$ and $\top$ operations, on the other hand, combine only the prohibitions/preferences of their operands, retaining only the preferences/prohibitions that they have in common. Finally, the $\neg$ operation inverts policy intent by swapping preferences and prohibitions.

The compositional approach to policy specification is largely about avoiding the authoring of additional standalone policies. Instead, it is preferable to repurpose existing policies via combination to achieve the desired result. We believe that our standard operations constitute the most practically useful combination approaches, and enable, through combinations of them, a rich set of composition strategies. The expressiveness of these operations is the subject of Section 4.5.

It is important that policies should be closed under composition so that new policies can be built from existing ones. Compositions should not behave differently from individual policies. Because PoCo policies output signed regular languages and their compositions employ the signed regular operations, $\mathfrak{R}^{\pm}$ should be closed under these operations. Theorem 3.1 proves that these operations, when applied to signed regular languages, result in signed regular languages.

Theorem 3.1 (*Closure of Signed Regular Languages*). Let $A = \langle A^+, A^- \rangle$ and $B = \langle B^+, B^- \rangle$ be signed regular languages. The following languages are signed regular:

1. $A \pm B$

   (a) Let $X = A^+ \cup B^+$ and $Y = A^- \cup B^-$ (By assumption)

   (b) A^+, A^-, B^+, B^- are regular (By def. of $\mathfrak{R}^{\pm}$)

   (c) $A \pm B = \langle X, Y \setminus X \rangle$ (By (a) and def. of $\pm$)

   (d) X and Y are regular (By (a), (b), and closure of $\mathfrak{R}$ under $\cup$)

   (e) $Y \setminus X$ is regular (By (d) and closure of $\mathfrak{R}$ under $\setminus$)

   (f) $X \cap (Y \setminus X) = \emptyset$ (Set algebra identity)

   (g) $\langle X, Y \setminus X \rangle$ is signed regular (By (d)–(f) and def. of $\mathfrak{R}^{\pm}$)

(h) Result follows from (c) and (g).

2. $A \mp B$

(i) $A \mp B = \langle X \smallsetminus Y, Y \rangle$ (By (a) and def. of $\pm$)

(j) $X \smallsetminus Y$ is regular (By (d) and closure of $\mathfrak{R}$ under $\smallsetminus$)

(k) $Y \cap (X \smallsetminus Y) = \emptyset$ (Set algebra identity)

(l) $\langle X \smallsetminus Y, Y \rangle$ is signed regular (By (d), (j), (k), and def. of $\mathfrak{R}^{\pm}$)

(m) Result follows from (i) and (l).

3. $A \perp B$

(n) $A \perp B = \langle A^+ \cap B^+, Y \rangle$ (By (a) and def. of $\perp$)

(o) $A^+ \cap B^+$ and Y are regular (By (a), (b), (d) and closure of $\mathfrak{R}$ under $\cap$)

(p) $A^+ \cap B^+ \cap Y =$
$(A^+ \cap B^+ \cap A^-) \cup (A^+ \cap B^+ \cap B^-) = \emptyset$ (By (a), distribution, and def. of $\mathfrak{R}^{\pm}$)

(q) $\langle A^+ \cap B^+, Y \rangle$ is signed regular (By (d), (o), (p), and def. of $\mathfrak{R}^{\pm}$)

(r) Result follows from (n) and (q).

4. $A \top B$

(s) $A \top B = \langle X, A^- \cap B^- \rangle$ (By (a) and def. of $\top$)

(t) X and $A^- \cap B^-$ are regular (By (a), (b), (d), and closure of $\mathfrak{R}$ under $\cap$)

(u) $X \cap A^- \cap B^- =$
$(A^+ \cap A^- \cap B^-) \cup (B^+ \cap A^- \cap B^-) = \emptyset$ (By (a), distribution, and def. of $\mathfrak{R}^{\pm}$)

(v) $\langle X, A^- \cap B^- \rangle$ is signed regular (By (d), (t), (u), and def. of $\mathfrak{R}^{\pm}$)

(w) Result follows from (s) and (v).

5. $\neg A$

(x) Proof is immediate from the definition of $\neg$ and that A^+ and A^- remain regular and disjoint as no operation is performed on them. $\square$

Now that we have a closed system of values and operations, we are equipped to investigate their properties in the following section.

4.4 Properties of Signed Regular Languages

We next evaluate the signed regular languages and operations by exploring the properties they satisfy. This collection of algebraic identities serves as both a toolbox for policy authors and a sanity check on the operation definitions. Some of these properties can be applied as optimizations, reducing the overall number of operations performed. Furthermore, these properties establish the signed regular languages under subsets of the operations as instances of various algebraic structures. This ties our system to results from abstract algebra, from which it can be further understood.

Theorem 4.2 (*Algebraic Properties of Signed Regular languages*). Let $A = \langle A^+, A^- \rangle$, $B = \langle B^+, B^- \rangle$, and $C = \langle C^+, C^- \rangle$ be signed regular languages. The following equivalences hold:

1. $\pm$ is commutative: $A \pm B = B \pm A$

   Proof: $A \pm B = B \pm A$

   $\equiv \langle A^+ \cup B^+, (A^- \cup B^-) \setminus (A^+ \cup B^+) \rangle = \langle B^+ \cup A^+, (B^- \cup A^-) \setminus (B^+ \cup A^+) \rangle$

   (Def. of $\pm$)

   $\equiv \langle A^+ \cup B^+, (A^- \cup B^-) \setminus (A^+ \cup B^+) \rangle = \langle A^+ \cup B^+, (A^- \cup B^-) \setminus (A^+ \cup B^+) \rangle$

   ($\cup$ commutative)

   $\square$

2. $\mp$ is commutative: $A \mp B = B \mp A$

   Proof: $A \mp B = B \mp A$

   $\equiv \langle (A^+ \cup B^+) \setminus (A^- \cup B^-), (A^- \cup B^-) \rangle = \langle (B^+ \cup A^+) \setminus (B^- \cup A^-), (B^- \cup A^-) \rangle$

   (Def. of $\pm$)

$\equiv \langle (A^+ \cup B^+) \setminus (A^- \cup B^-), (A^- \cup B^-) \rangle = \langle (A^+ \cup B^+) \setminus (A^- \cup B^-), (A^- \cup B^-) \rangle$

($\cup$ commutative)

□

3. $\perp$ is commutative: $A \perp B = B \perp A$

   Proof: $A \perp B = B \perp A$

   $\equiv \langle A^+ \cap B^+, A^- \cup B^- \rangle = \langle B^+ \cap A^+, B^- \cup A^- \rangle$ **(Def. of $\perp$)**

   $\equiv \langle A^+ \cap B^+, A^- \cup B^- \rangle = \langle A^+ \cap B^+, A^- \cup B^- \rangle$ **($\cup, \cap$ commutative)**

   □

4. $\top$ is commutative: $A \top B = B \top A$

   Proof: $A \top B = B \top A$

   $\equiv \langle A^+ \cup B^+, A^- \cap B^- \rangle = \langle B^+ \cup A^+, B^- \cap A^- \rangle$ **(Def. of $\top$)**

   $\equiv \langle A^+ \cup B^+, A^- \cap B^- \rangle = \langle A^+ \cup B^+, A^- \cap B^- \rangle$ **($\cup, \cap$ commutative)**

   □

5. $\pm$ is associative: $A \pm (B \pm C) = (A \pm B) \pm C$

   Proof: $A \pm (B \pm C) = (A \pm B) \pm C$

   $\equiv \langle A^+ \cup (B^+ \cup C^+), (A^- \cup ((B^- \cup C^-) \setminus (B^+ \cup C^+))) \setminus (A^+ \cup (B^+ \cup C^+)) \rangle =$

   $\langle (A^+ \cup B^+) \cup C^+, (((A^- \cup B^-) \setminus (A^+ \cup B^+)) \cup C^-) \setminus ((A^+ \cup B^+) \cup C^+) \rangle$**(Def. of $\pm$)**

   $\equiv \langle A^+ \cup (B^+ \cup C^+), (A^- \cup ((B^- \cup C^-) \setminus (B^+ \cup C^+))) \setminus (A^+ \cup (B^+ \cup C^+)) \rangle =$

   $\langle A^+ \cup (B^+ \cup C^+), (((A^- \cup B^-) \setminus (A^+ \cup B^+)) \cup C^-) \setminus (A^+ \cup (B^+ \cup C^+)) \rangle$

   ($\cup$ associative)

   $\equiv A^- \cup ((B^- \cup C^-) \setminus (B^+ \cup C^+))) \setminus (A^+ \cup (B^+ \cup C^+)) =$

   $(((A^- \cup B^-) \setminus (A^+ \cup B^+)) \cup C^-) \setminus (A^+ \cup (B^+ \cup C^+))$

   (First part of tuple finished, omit)

   $\equiv (A^- \cup ((B^- \cup C^-) \cap \overline{(B^+ \cup C^+)})) \cap \overline{(A^+ \cup (B^+ \cup C^+))} =$

   $(((A^- \cup B^-) \cap \overline{(A^+ \cup B^+)}) \cup C^-) \cap \overline{(A^+ \cup (B^+ \cup C^+))}$ **($X \setminus Y = X \cap \overline{Y}$)**

   $\equiv (A^- \cup ((B^- \cap \overline{(B^+ \cup C^+)}) \cup (C^- \cap \overline{(B^+ \cup C^+)}))) \cap \overline{(A^+ \cup (B^+ \cup C^+))} =$

$(((A^- \cap \overline{(A^+ \cup B^+)}) \cup (B^- \cap \overline{(A^+ \cup B^+)})) \cup C^-) \cap \overline{(A^+ \cup (B^+ \cup C^+))}$**(Distribute)**

$\equiv (A^- \cup ((B^- \cap (\overline{B^+} \cap \overline{C^+})) \cup (C^- \cap (\overline{B^+} \cap \overline{C^+})))) \cap (\overline{A^+} \cap (\overline{B^+} \cap \overline{C^+})) =$

$(((A^- \cap (\overline{A^+} \cap \overline{B^+})) \cup (B^- \cap (\overline{A^+} \cap \overline{B^+}))) \cup C^-) \cap (\overline{A^+} \cap (\overline{B^+} \cap \overline{C^+}))$**(**$\overline{X \cup Y} = \overline{X} \cap \overline{Y}$**)**

$\equiv (A^- \cup (B^- \cap \overline{C^+}) \cup (C^- \cap \overline{B^+})) \cap (\overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) =$

$((A^- \cap \overline{B^+}) \cup (B^- \cap \overline{A^+}) \cup C^-) \cap (\overline{A^+} \cap \overline{B^+} \cap \overline{C^+})$ **(**$X^- \cap \overline{X^+} = X^-$**)**

$\equiv (A^- \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) \cup (B^- \cap \overline{C^+} \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) \cup (C^- \cap \overline{B^+} \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) =$

$(A^- \cap \overline{B^+} \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) \cup (B^- \cap \overline{A^+} \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) \cup (C^- \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+})$

(Distribute)

$\equiv (A^- \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) \cup (B^- \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) \cup (C^- \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) =$

$(A^- \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) \cup (B^- \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) \cup (C^- \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+})$**(**$X \cap X = X$**)**

□

6. $\mp$ is associative: $A \mp (B \mp C) = (A \mp B) \mp C$

   Proof: $A \mp (B \mp C) = (A \mp B) \mp C$

   $\equiv \langle (A^+ \cup ((B^+ \cup C^+) \setminus (B^- \cup C^-))) \setminus (A^- \cup (B^- \cup C^-)), A^- \cup (B^- \cup C^-) \rangle =$

   $\langle (((A^+ \cup B^+) \setminus (A^- \cup B^-)) \cup C^+) \setminus ((A^- \cup B^-) \cup C^-), (A^- \cup B^-) \cup C^- \rangle$**(Def. of** $\mp$**)**

   $\equiv \langle (A^+ \cup ((B^+ \cup C^+) \setminus (B^- \cup C^-))) \setminus (A^- \cup (B^- \cup C^-)), (A^- \cup B^-) \cup C^- \rangle =$

   $\langle (((A^+ \cup B^+) \setminus (A^- \cup B^-)) \cup C^+) \setminus ((A^- \cup B^-) \cup C^-), (A^- \cup B^-) \cup C^- \rangle$

   ($\cup$ **associative)**

   $\equiv (A^+ \cup ((B^+ \cup C^+) \setminus (B^- \cup C^-))) \setminus (A^- \cup (B^- \cup C^-)) =$

   $(((A^+ \cup B^+) \setminus (A^- \cup B^-)) \cup C^+) \setminus ((A^- \cup B^-) \cup C^-)$

   (Second part of tuple finished, omit)

   $\equiv (A^+ \cup ((B^+ \cup C^+) \cap \overline{(B^- \cup C^-)})) \cap \overline{(A^- \cup B^- \cup C^-)} =$

   $(((A^+ \cup B^+) \cap \overline{(A^- \cup B^-)}) \cup C^+) \cap \overline{(A^- \cup B^- \cup C^-)}$ **(**$X \setminus Y = X \cap \overline{Y}$**)**

   $\equiv (A^+ \cup ((B^+ \cap \overline{(B^- \cup C^-)}) \cup (C^+ \cap \overline{(B^- \cup C^-)}))) \cap \overline{(A^- \cup B^- \cup C^-)} =$

   $(((A^+ \cap \overline{(A^- \cup B^-)}) \cup (B^+ \cap \overline{(A^- \cup B^-)})) \cup C^+) \cap \overline{(A^- \cup B^- \cup C^-)}$ **(Distribute)**

   $\equiv (A^+ \cup (B^+ \cap \overline{B^-} \cap \overline{C^-}) \cup (C^+ \cap \overline{B^-} \cap \overline{C^-})) \cap (\overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) =$

$((A^+ \cap \overline{A^-} \cap \overline{B^-}) \cup (B^+ \cap \overline{A^-} \cap \overline{B^-}) \cup C^+) \cap (\overline{A^-} \cap \overline{B^-} \cap \overline{C^-})$ $(\overline{X \cup Y} = \overline{X} \cap \overline{Y})$

$\equiv (A^+ \cup (B^+ \cap \overline{C^-}) \cup (C^+ \cap \overline{B^-})) \cap (\overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) =$

$((A^+ \cap \overline{B^-}) \cup (B^+ \cap \overline{A^-}) \cup C^+) \cap (\overline{A^-} \cap \overline{B^-} \cap \overline{C^-})$ $(X^+ \cap \overline{X^-} = X^+)$

$\equiv (A^+ \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) \cup (B^+ \cap \overline{C^-} \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) \cup (C^+ \cap \overline{B^-} \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) =$

$(A^+ \cap \overline{B^-} \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) \cup (B^+ \cap \overline{A^-} \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) \cup (C^+ \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-})$

(Distribute)

$\equiv (A^+ \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) \cup (B^+ \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) \cup (C^+ \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) =$

$(A^+ \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) \cup (B^+ \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) \cup (C^+ \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-})(X \cup X = X)$

□

7. $\perp$ is associative: $A \perp (B \perp C) = (A \perp B) \perp C$

   Proof: $A \perp (B \perp C) = (A \perp B) \perp C$

   $\equiv \langle A^+, A^- \rangle \perp \langle B^+ \cap C^+, B^- \cup C^- \rangle = \langle A^+ \cap B^+, A^- \cup B^- \rangle \perp \langle C^+, C^- \rangle$ **(Def. of $\perp$)**

   $\equiv \langle A^+ \cap B^+ \cap C^+, A^- \cup B^- \cup C^- \rangle = \langle A^+ \cap B^+ \cap C^+, A^- \cup B^- \cup C^- \rangle$**(Def. of $\perp$)**

   □

8. $\top$ is associative: $A \top (B \top C) = (A \top B) \top C$

   Proof: $A \top (B \top C) = (A \top B) \top C$

   $\equiv \langle A^+, A^- \rangle \top \langle B^+ \cup C^+, B^- \cap C^- \rangle = \langle A^+ \cup B^+, A^- \cap B^- \rangle \top \langle C^+, C^- \rangle$ **(Def. of $\top$)**

   $\equiv \langle A^+ \cup B^+ \cup C^+, A^- \cap B^- \cap C^- \rangle = \langle A^+ \cup B^+ \cup C^+, A^- \cap B^- \cap C^- \rangle$**(Def. of $\top$)**

   □

9. $\neg$ is involutary: $\neg(\neg A) = A$

   Proof: $\neg(\neg A) = A$

   $\equiv \neg\langle A^-, A^+ \rangle = \langle A^+, A^- \rangle$ **(Def. of $\neg$)**

   $\equiv \langle A^+, A^- \rangle = \langle A^+, A^- \rangle$ **(Def. of $\neg$)**

   □

10. De Morgan's Law ($\pm$): $\neg(A \pm B) = (\neg A) \mp (\neg B)$

    Proof: $\neg(A \pm B) = (\neg A) \mp (\neg B)$

    $\equiv \neg\langle A^+ \cup B^+, (A^- \cup B^-) \setminus (A^+ \cup B^+) \rangle = (\neg\langle A^+, A^- \rangle) \mp (\neg\langle B^+, B^- \rangle)$**(Def. of $\pm$)**

$\equiv \langle (A^- \cup B^-) \setminus (A^+ \cup B^+), A^+ \cup B^+ \rangle = \langle A^-, A^+ \rangle \mp \langle B^-, B^+ \rangle$ **(Def. of $\neg$)**

$\equiv \langle (A^- \cup B^-) \setminus (A^+ \cup B^+), A^+ \cup B^+ \rangle = \langle (A^- \cup B^-) \setminus (A^+ \cup B^+), A^+ \cup B^+ \rangle$

(Def. of $\mp$)

□

11. De Morgan's Law ($\mp$): $\neg(A \mp B) = (\neg A) \pm (\neg B)$

Proof: $\neg(A \mp B) = (\neg A) \pm (\neg B)$

$\equiv \neg\langle (A^+ \cup B^+) \setminus (A^- \cup B^-), A^- \cup B^- \rangle = (\neg\langle A^+, A^- \rangle) \pm (\neg\langle B^+, B^- \rangle)$ **(Def. of $\mp$)**

$\equiv \langle A^- \cup B^-, (A^+ \cup B^+) \setminus (A^- \cup B^-) \rangle = \langle A^-, A^+ \rangle \pm \langle B^-, B^+ \rangle$ **(Def. of $\neg$)**

$\equiv \langle A^- \cup B^-, (A^+ \cup B^+) \setminus (A^- \cup B^-) \rangle = \langle A^- \cup B^-, (A^+ \cup B^+) \setminus (A^- \cup B^-) \rangle$

(Def. of $\pm$)

□

12. De Morgan's Law ($\bot$): $\neg(A \bot B) = (\neg A) \top (\neg B)$

Proof: $\equiv \neg\langle A^+ \cap B^+, A^- \cup B^- \rangle = (\neg\langle A^+, A^- \rangle) \top (\neg\langle B^+, B^- \rangle)$ **(Def. of $\bot$)**

$\equiv \langle A^- \cup B^-, A^+ \cap B^+ \rangle = \langle A^-, A^+ \rangle \top \langle B^-, B^+ \rangle$ **(Def. of $\neg$)**

$\equiv \langle A^- \cup B^-, A^+ \cap B^+ \rangle = \langle A^- \cup B^-, A^+ \cap B^+ \rangle$ **(Def. of $\top$)**

□

13. De Morgan's Law ($\top$): $\neg(A \top B) = (\neg A) \bot (\neg B)$

Proof: $\neg(A \top B) = (\neg A) \bot (\neg B)$

$\equiv \neg\langle A^+ \cup B^+, A^- \cap B^- \rangle = (\neg\langle A^+, A^- \rangle) \bot (\neg\langle B^+, B^- \rangle)$ **(Def. of $\top$)**

$\equiv \langle A^- \cap B^-, A^+ \cup B^+ \rangle = \langle A^-, A^+ \rangle \bot \langle B^-, B^+ \rangle$ **(Def. of $\neg$)**

$\equiv \langle A^- \cap B^-, A^+ \cup B^+ \rangle = \langle A^- \cap B^-, A^+ \cup B^+ \rangle$ **(Def. of $\bot$)**

□

14. $\pm = \mp$ for non-conflicting operands: $(A^+ \cap B^- = \emptyset \wedge B^+ \cap A^- = \emptyset) \Rightarrow (A \mp B = A \pm B)$

Proof: $(A \mp B = A \pm B)$

$\equiv \langle (A^+ \cup B^+) \setminus (A^- \cup B^-), (A^- \cup B^-) \rangle = \langle A^+ \cup B^+, (A^- \cup B^-) \setminus (A^+ \cup B^+) \rangle$

(Def. of $\mp, \pm$)

$\equiv \langle (A^+ \cup B^+) \cap \overline{(A^- \cup B^-)}, (A^- \cup B^-) \rangle = \langle A^+ \cup B^+, (A^- \cup B^-) \cap \overline{(A^+ \cup B^+)} \rangle$

$(X \setminus Y = X \cap \overline{Y})$

$\equiv \langle (A^+ \cup B^+) \cap \overline{A^-} \cap \overline{B^-}, (A^- \cup B^-) \rangle = \langle A^+ \cup B^+, (A^- \cup B^-) \cap \overline{A^+} \cap \overline{B^+} \rangle (\overline{X \cup Y} = \overline{X} \cap \overline{Y})$

$\equiv \langle (A^+ \cap \overline{A^-} \cap \overline{B^-}) \cup (B^+ \cap \overline{A^-} \cap \overline{B^-}), A^- \cup B^- \rangle =$

$\langle A^+ \cup B^+, (A^- \cap \overline{A^+} \cap \overline{B^+}) \cup (B^- \cap \overline{A^+} \cap \overline{B^+}) \rangle$ **(Distribute)**

$\equiv \langle A^+ \cup B^+, A^- \cup B^- \rangle = \langle A^+ \cup B^+, A^- \cup B^- \rangle$ $(X \cap Y = \emptyset \Rightarrow X \cap \overline{Y} = X)$

□

15. θ is an identity element for $\mp$: $A \mp \theta = A$

Proof: $A \mp \theta = A$

$\equiv \langle (A^+ \cup \emptyset) \setminus (A^- \cup \emptyset), A^- \cup \emptyset \rangle = \langle A^+, A^- \rangle$ **(Def. of $\mp$)**

$\equiv \langle A^+ \setminus A^-, A^- \rangle = \langle A^+, A^- \rangle$ $(X \cup \emptyset = X)$

$\equiv \langle A^+, A^- \rangle = \langle A^+, A^- \rangle$ $(A^+ \cap A^- = \emptyset \Rightarrow A^+ \setminus A^- = A^+)$

□

16. θ is an identity element for $\pm$: $A \pm \theta = A$

This result follows from Theorem 4.2.14 and Theorem 4.2.15.

17. $\langle \Sigma^+, \emptyset \rangle$ is an identity element for $\perp$: $\langle \Sigma^+, \emptyset \rangle \perp A = A$

Proof: $\langle \Sigma^+, \emptyset \rangle \perp A = A$

$\equiv \langle \Sigma^+ \cap A^+, \emptyset \cup A^- \rangle = \langle A^+, A^- \rangle$ **(Def. of $\perp$)**

$\equiv \langle A^+, A^- \rangle = \langle A^+, A^- \rangle$ $(A^+ \subseteq \Sigma^+, \emptyset \cup X = X)$

□

18. $\langle \emptyset, \Sigma^+ \rangle$ is an identity element for $\top$: $\langle \emptyset, \Sigma^+ \rangle \top A = A$

Proof: $\langle \emptyset, \Sigma^+ \rangle \top A = A$

$\equiv \langle \emptyset \cup A^+, \Sigma^+ \cap A^- \rangle \top = \langle A^+, A^- \rangle$ **(Def. of $\top$)**

$\equiv \langle A^+, A^- \rangle = \langle A^+, A^- \rangle$ $(\emptyset \cup X = X, A^- \subseteq \Sigma^+)$

□

19. $\langle \emptyset, \Sigma^+ \rangle$ is an absorbing element for $\perp$: $\langle \emptyset, \Sigma^+ \rangle \perp A = \langle \emptyset, \Sigma^+ \rangle$

Proof: $\langle \emptyset, \Sigma^+ \rangle \perp A = \langle \emptyset, \Sigma^+ \rangle$

$\equiv \langle \emptyset \cap A^+, \Sigma^+ \cup A^- \rangle = \langle \emptyset, \Sigma^+ \rangle$ **(Def. of $\perp$)**

$\equiv \langle \emptyset, \Sigma^+ \rangle = \langle \emptyset, \Sigma^+ \rangle$ $(\emptyset \cap X = \emptyset, A^- \subseteq \Sigma^+)$

□

20. $\langle \Sigma^+, \emptyset \rangle$ is an absorbing element for $\top$: $\langle \Sigma^+, \emptyset \rangle \top A = \langle \Sigma^+, \emptyset \rangle$

Proof: $\langle \Sigma^+, \emptyset \rangle \top A = \langle \Sigma^+, \emptyset \rangle$

$\equiv \langle \Sigma^+ \cup A^+, \emptyset \cap A^- \rangle = \langle \Sigma^+, \emptyset \rangle$ **(Def. of $\top$)**

$\equiv \langle \Sigma^+, \emptyset \rangle = \langle \Sigma^+, \emptyset \rangle$ $(A^+ \subseteq \Sigma^+, \emptyset \cap X = \emptyset)$

□

21. $\pm$ is idempotent: $A \pm A = A$

Proof: $A \pm A = A$

$\equiv \langle A^+ \cup A^+, (A^- \cup A^-) \setminus (A^+ \cup A^+) \rangle = \langle A^+, A^- \rangle$ **(Def. of $\pm$)**

$\equiv \langle A^+, A^- \setminus A^+ \rangle = \langle A^+, A^- \rangle$ $(X \cup X = X)$

$\equiv \langle A^+, A^- \rangle = \langle A^+, A^- \rangle$ $(X \cap Y = \emptyset \Rightarrow (X \setminus Y = X))$

□

22. $\mp$ is idempotent: $A \mp A = A$

Proof: $A \mp A = A$

$\equiv \langle (A^+ \cup A^+) \setminus (A^- \cup A^-), A^- \cup A^- \rangle = \langle A^+, A^- \rangle$ **(Def. of $\mp$)**

$\equiv \langle A^+ \setminus A^-, A^- \rangle = \langle A^+, A^- \rangle$ $(X \cup X = X)$

$\equiv \langle A^+, A^- \rangle = \langle A^+, A^- \rangle$ $(X \cap Y = \emptyset \Rightarrow (X \setminus Y = X))$

□

23. $\top$ is idempotent: $A \top A = A$

Proof: $A \top A = A$

$\equiv \langle A^+ \cup A^+, A^- \cap A^- \rangle = \langle A^+, A^- \rangle$ **(Def. of $\top$)**

$\equiv \langle A^+, A^- \rangle = \langle A^+, A^- \rangle$ $(X \cup X = X, Y \cap Y = Y)$

□

24. $\perp$ is idempotent: $A \perp A = A$

Proof: $A \perp A = A$

$\equiv \langle A^+ \cap A^+, A^- \cup A^- \rangle = \langle A^+, A^- \rangle$ **(Def. of $\perp$)**

$\equiv \langle A^+, A^- \rangle = \langle A^+, A^- \rangle$ $(X \cup X = X, Y \cap Y = Y)$

$\square$

25. $\perp$ distributes over $\mp$: $A \perp (B \mp C) = (A \perp B) \mp (A \perp C)$

Proof: $A \perp (B \mp C) = (A \perp B) \mp (A \perp C)$

$\equiv \langle A^+, A^- \rangle \perp \langle (B^+ \cup C^+) \setminus (B^- \cup C^-), B^- \cup C^- \rangle =$

$(\langle A^+, A^- \rangle \perp \langle B^+, B^- \rangle) \mp (\langle A^+, A^- \rangle \perp \langle C^+, C^- \rangle)$ **(Def. of $\mp$)**

$\equiv \langle A^+, A^- \rangle \perp \langle (B^+ \cup C^+) \cap \overline{(B^- \cup C^-)}, B^- \cup C^- \rangle =$

$(\langle A^+, A^- \rangle \perp \langle B^+, B^- \rangle) \mp (\langle A^+, A^- \rangle \perp \langle C^+, C^- \rangle)$ $(X \setminus Y = X \cap \overline{Y})$

$\equiv \langle A^+, A^- \rangle \perp \langle (B^+ \cup C^+) \cap (\overline{B^-} \cap \overline{C^-}), B^- \cup C^- \rangle =$

$(\langle A^+, A^- \rangle \perp \langle B^+, B^- \rangle) \mp (\langle A^+, A^- \rangle \perp \langle C^+, C^- \rangle)$ $(\overline{X \cup Y} = \overline{X} \cap \overline{Y})$

$\equiv \langle A^+, A^- \rangle \perp \langle (B^+ \cap \overline{B^-} \cap \overline{C^-}) \cup (C^+ \cap \overline{B^-} \cap \overline{C^-}), B^- \cup C^- \rangle =$

$(\langle A^+, A^- \rangle \perp \langle B^+, B^- \rangle) \mp (\langle A^+, A^- \rangle \perp \langle C^+, C^- \rangle)$ **(Distribute)**

$\equiv \langle A^+, A^- \rangle \perp \langle (B^+ \cap \overline{C^-}) \cup (C^+ \cap \overline{B^-}), B^- \cup C^- \rangle =$

$(\langle A^+, A^- \rangle \perp \langle B^+, B^- \rangle) \mp (\langle A^+, A^- \rangle \perp \langle C^+, C^- \rangle)$ $(X^+ \cap \overline{X^-} = X^+)$

$\equiv \langle A^+ \cap ((B^+ \cap \overline{C^-}) \cup (C^+ \cap \overline{B^-})), A^- \cup B^- \cup C^- \rangle =$

$\langle A^+ \cap B^+, A^- \cup B^- \rangle \mp \langle A^+ \cap C^+, A^- \cup C^- \rangle$ **(Def. of $\perp$)**

$\equiv \langle (A^+ \cap B^+ \cap \overline{C^-}) \cup (A^+ \cap C^+ \cap \overline{B^-}), A^- \cup B^- \cup C^- \rangle =$

$\langle A^+ \cap B^+, A^- \cup B^- \rangle \mp \langle A^+ \cap C^+, A^- \cup C^- \rangle$ **(Distribute)**

$\equiv \langle (A^+ \cap B^+ \cap \overline{C^-}) \cup (A^+ \cap C^+ \cap \overline{B^-}), A^- \cup B^- \cup C^- \rangle =$

$\langle ((A^+ \cap B^+) \cup (A^+ \cap C^+)) \setminus (A^- \cup B^- \cup A^- \cup C^-), A^- \cup B^- \cup A^- \cup C^- \rangle$ **(Def of $\mp$)** $\equiv \langle (A^+ \cap B^+ \cap \overline{C^-}) \cup (A^+ \cap C^+ \cap \overline{B^-}), A^- \cup B^- \cup C^- \rangle =$

$\langle ((A^+ \cap B^+) \cup (A^+ \cap C^+)) \setminus (A^- \cup B^- \cup C^-), A^- \cup B^- \cup C^- \rangle$ $(X \cup X = X)$

$\equiv (A^+ \cap B^+ \cap \overline{C^-}) \cup (A^+ \cap C^+ \cap \overline{B^-}) = ((A^+ \cap B^+) \cup (A^+ \cap C^+)) \setminus (A^- \cup B^- \cup C^-)$

(Second part of tuple finished, omit)

$\equiv (A^+ \cap B^+ \cap \overline{C^-}) \cup (A^+ \cap C^+ \cap \overline{B^-}) = ((A^+ \cap B^+) \cup (A^+ \cap C^+)) \cap \overline{(A^- \cup B^- \cup C^-)}$

$(X \setminus Y = X \cap \overline{Y})$

$\equiv (A^+ \cap B^+ \cap \overline{C^-}) \cup (A^+ \cap C^+ \cap \overline{B^-}) =$

$((A^+ \cap B^+ \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}) \cup (A^+ \cap C^+ \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}))$ **(Distribute)**

$\equiv (A^+ \cap B^+ \cap \overline{C^-}) \cup (A^+ \cap C^+ \cap \overline{B^-}) = (A^+ \cap B^+ \cap \overline{C^-}) \cup (A^+ \cap C^+ \cap \overline{B^-})$

$(X^+ \cap \overline{X^-} = X^+)$

□

26. $\mp$ distributes over $\perp$: $A \mp (B \perp C) = (A \mp B) \perp (A \mp C)$

Proof: $A \mp (B \perp C) = (A \mp B) \perp (A \mp C)$

$\equiv \langle A^+, A^- \rangle \mp \langle B^+ \cap C^+, B^- \cup C^- \rangle = (\langle A^+, A^- \rangle \mp \langle B^+, B^- \rangle) \perp (\langle A^+, A^- \rangle \mp \langle C^+, C^- \rangle)$

(Def. of $\perp$)

$\equiv \langle (A^+ \cup (B^+ \cap C^+)) \setminus (A^- \cup B^- \cup C^-), A^- \cup B^- \cup C^- \rangle =$

$\langle (A^+ \cup B^+) \setminus (A^- \cup B^-), A^- \cup B^- \rangle \perp \langle (A^+ \cup C^+) \setminus (A^- \cup C^-), A^- \cup C^- \rangle$

(Def. of $\mp$)

$\equiv \langle (A^+ \cup (B^+ \cap C^+)) \setminus (A^- \cup B^- \cup C^-), A^- \cup B^- \cup C^- \rangle =$

$\langle ((A^+ \cup B^+) \setminus (A^- \cup B^-)) \cap ((A^+ \cup C^+) \setminus (A^- \cup C^-)), A^- \cup B^- \cup A^- \cup C^- \rangle$

(Def. of $\perp$)

$\equiv \langle (A^+ \cup (B^+ \cap C^+)) \setminus (A^- \cup B^- \cup C^-), A^- \cup B^- \cup C^- \rangle =$

$\langle ((A^+ \cup B^+) \setminus (A^- \cup B^-)) \cap ((A^+ \cup C^+) \setminus (A^- \cup C^-)), A^- \cup B^- \cup C^- \rangle$

$(X \cup X = X)$

$\equiv (A^+ \cup (B^+ \cap C^+)) \setminus (A^- \cup B^- \cup C^-) =$

$((A^+ \cup B^+) \setminus (A^- \cup B^-)) \cap ((A^+ \cup C^+) \setminus (A^- \cup C^-))$

(Second part of tuple finished, omit)

$\equiv (A^+ \cup (B^+ \cap C^+)) \cap \overline{(A^- \cup B^- \cup C^-)} =$

$((A^+ \cup B^+) \cap \overline{(A^- \cup B^-)}) \cap ((A^+ \cup C^+) \cap \overline{(A^- \cup C^-)})$ $(X \setminus Y = X \cap \overline{Y})$

$\equiv (A^+ \cup (B^+ \cap C^+)) \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-} = (A^+ \cup B^+) \cap \overline{A^-} \cap \overline{B^-} \cap (A^+ \cup C^+) \cap \overline{A^-} \cap \overline{C^-}$

$(\overline{X \cup Y} = \overline{X} \cap \overline{Y})$

$\equiv (A^+ \cup (B^+ \cap C^+)) \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-} = (A^+ \cup B^+) \cap \overline{A^-} \cap \overline{B^-} \cap (A^+ \cup C^+) \cap \overline{C^-}$

($X \cap X = X$)

$\equiv (A^+ \cup B^+) \cap (A^+ \cup C^+) \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-} = (A^+ \cup B^+) \cap (A^+ \cup C^+) \cap \overline{A^-} \cap \overline{B^-} \cap \overline{C^-}$

(Distribute, $\cap$ commutative)

□

27. $\top$ distributes over $\pm$: $A \top (B \pm C) = (A \top B) \pm (A \top C)$

Proof: $A \top (B \pm C) = (A \top B) \pm (A \top C)$

$\equiv \langle A^+, A^- \rangle \top (\langle B^+ \cup C^+, (B^- \cup C^-) \setminus (B^+ \cup C^+) \rangle) =$

$(\langle A^+, A^- \rangle \top \langle B^+, B^- \rangle) \pm (\langle A^+, A^- \rangle \top \langle C^+, C^- \rangle)$ **(Def. of $\pm$)**

$\equiv \langle A^+ \cup B^+ \cup C^+, A^- \cap ((B^- \cup C^-) \setminus (B^+ \cup C^+)) \rangle =$

$\langle A^+ \cup B^+, A^- \cap B^- \rangle \pm \langle A^+ \cup C^+, A^- \cap C^- \rangle$ **(Def. of $\top$)**

$\equiv \langle A^+ \cup B^+ \cup C^+, A^- \cap ((B^- \cup C^-) \setminus (B^+ \cup C^+)) \rangle =$

$\langle A^+ \cup B^+ \cup A^+ \cup C^+, ((A^- \cap B^-) \cup (A^- \cap C^-)) \setminus (A^+ \cup B^+ \cup A^+ \cup C^+) \rangle$

(Def. of $\pm$)

$\equiv \langle A^+ \cup B^+ \cup C^+, A^- \cap ((B^- \cup C^-) \setminus (B^+ \cup C^+)) \rangle =$

$\langle A^+ \cup B^+ \cup C^+, ((A^- \cap B^-) \cup (A^- \cap C^-)) \setminus (A^+ \cup B^+ \cup C^+) \rangle$ **($X \cup X = X$)**

$\equiv A^- \cap ((B^- \cup C^-) \setminus (B^+ \cup C^+)) = ((A^- \cap B^-) \cup (A^- \cap C^-)) \setminus (A^+ \cup B^+ \cup C^+)$

(First part of tuple finished, omit)

$\equiv A^- \cap (B^- \cup C^-) \cap \overline{(B^+ \cup C^+)} = ((A^- \cap B^-) \cup (A^- \cap C^-)) \cap \overline{(A^+ \cup B^+ \cup C^+)}$

($X \setminus Y = X \cap \overline{Y}$)

$\equiv A^- \cap \overline{B^+} \cap \overline{C^+} \cap (B^- \cup C^-) = ((A^- \cap B^-) \cup (A^- \cap C^-)) \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}$

($\overline{X \cup Y} = \overline{X} \cap \overline{Y}$, $\cap$ commutative)

$\equiv (A^- \cap \overline{B^+} \cap \overline{C^+} \cap B^-) \cup (A^- \cap \overline{B^+} \cap \overline{C^+} \cap C^-) =$

$(A^- \cap B^- \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}) \cup (A^- \cap C^- \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+})$ **(Distribute)**

$\equiv (A^- \cap B^- \cap \overline{C^+}) \cup (A^- \cap \overline{B^+} \cap C^-) = (A^- \cap B^- \cap \overline{C^+}) \cup (A^- \cap \overline{B^+} \cap C^-)$

($\overline{X^+} \cap X^- = X^-$, $\cap$ commutative)

□

28. $\pm$ distributes over $\top$: $A \pm (B \top C) = (A \pm B) \top (A \pm C)$

Proof: $A \pm (B \top C) = (A \pm B) \top (A \pm C)$

$\equiv \langle A^+, A^- \rangle \pm \langle B^+ \cup C^+, B^- \cap C^- \rangle = (\langle A^+, A^- \rangle \pm \langle B^+, B^- \rangle) \top (\langle A^+, A^- \rangle \pm \langle C^+, C^- \rangle)$

(Def. of $\top$)

$\equiv \langle A^+ \cup B^+ \cup C^+, (A^- \cup (B^- \cap C^-)) \smallsetminus (A^+ \cup B^+ \cup C^+) \rangle =$

$\langle A^+ \cup B^+, (A^- \cup B^-) \smallsetminus (A^+ \cup B^+) \rangle \top \langle A^+ \cup C^+, (A^- \cup C^-) \smallsetminus (A^+ \cup C^+) \rangle$**(Def. of $\pm$)**

$\equiv \langle A^+ \cup B^+ \cup C^+, (A^- \cup (B^- \cap C^-)) \smallsetminus (A^+ \cup B^+ \cup C^+) \rangle =$

$\langle A^+ \cup B^+ \cup A^+ \cup C^+, ((A^- \cup B^-) \smallsetminus (A^+ \cup B^+)) \cap ((A^- \cup C^-) \smallsetminus (A^+ \cup C^+)) \rangle$

(Def. of $\top$)

$\equiv \langle A^+ \cup B^+ \cup C^+, (A^- \cup (B^- \cap C^-)) \smallsetminus (A^+ \cup B^+ \cup C^+) \rangle =$

$\langle A^+ \cup B^+ \cup C^+, ((A^- \cup B^-) \smallsetminus (A^+ \cup B^+)) \cap ((A^- \cup C^-) \smallsetminus (A^+ \cup C^+)) \rangle$**($X \cup X = X$)**

$\equiv (A^- \cup (B^- \cap C^-)) \smallsetminus (A^+ \cup B^+ \cup C^+) =$

$((A^- \cup B^-) \smallsetminus (A^+ \cup B^+)) \cap ((A^- \cup C^-) \smallsetminus (A^+ \cup C^+))$

(First part of tuple finished, omit)

$\equiv (A^- \cup (B^- \cap C^-)) \cap \overline{(A^+ \cup B^+ \cup C^+)} =$

$((A^- \cup B^-) \cap \overline{(A^+ \cup B^+)}) \cap ((A^- \cup C^-) \cap \overline{(A^+ \cup C^+)})$ **($X \smallsetminus Y = X \cap \overline{Y}$)**

$\equiv (A^- \cup (B^- \cap C^-)) \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+} = (A^- \cup B^-) \cap (A^- \cup C^-) \cap \overline{A^+} \cap \overline{B^+} \cap \overline{A^+} \cap \overline{C^+}$

($\overline{X \cup Y} = \overline{X} \cap \overline{Y}$, $\cap$ commutative)

$\equiv (A^- \cup B^-) \cap (A^- \cup C^-) \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+} = (A^- \cup B^-) \cap (A^- \cup C^-) \cap \overline{A^+} \cap \overline{B^+} \cap \overline{C^+}$

($X \cap X = X$,Distribute)

□

29. Absorption for $\top$ over $\perp$: $A \top (A \perp B) = A$

Proof: $A \top (A \perp B) = A$

$\equiv \langle A^+, A^- \rangle \top \langle A^+ \cap B^+, A^- \cup B^- \rangle = \langle A^+, A^- \rangle$ **(Def. of $\perp$)**

$\equiv \langle A^+ \cup (A^+ \cap B^+), A^- \cap (A^- \cup B^-) \rangle = \langle A^+, A^- \rangle$ **(Def. of $\top$)**

$\equiv \langle A^+, A^- \cap (A^- \cup B^-) \rangle = \langle A^+, A^- \rangle$ **($Y \subseteq X \Rightarrow X \cup Y = X$)**

$\equiv \langle A^+, A^- \rangle = \langle A^+, A^- \rangle$ $(X \subseteq Y \Rightarrow X \cap Y = X)$

□

30. Absorption for $\perp$ over $\top$: $A \perp (A \top B) = A$

Proof: $A \perp (A \top B) = A$

$\equiv \langle A^+, A^- \rangle \perp \langle A^+ \cup B^+, A^- \cap B^- \rangle = \langle A^+, A^- \rangle$ **(Def. of $\top$)**

$\equiv \langle A^+ \cap (A^+ \cup B^+), A^- \cup (A^- \cap B^-) \rangle = \langle A^+, A^- \rangle$ **(Def. of $\perp$)**

$\equiv \langle A^+, A^- \cup (A^- \cap B^-) \rangle = \langle A^+, A^- \rangle$ $(X \subseteq Y \Rightarrow X \cap Y = X)$

$\equiv \langle A^+, A^- \cup (A^- \cap B^-) \rangle = \langle A^+, A^- \rangle$ $(Y \subseteq X \Rightarrow X \cup Y = X)$

□

31. $\top$ distributes over $\perp$: $A \top (B \perp C) = (A \top B) \perp (A \top C)$

Proof: $A \top (B \perp C) = (A \top B) \perp (A \top C)$

$\equiv \langle A^+, A^- \rangle \top \langle B^+ \cap C^+, B^- \cup C^- \rangle = (\langle A^+, A^- \rangle \top \langle B^+, B^- \rangle) \perp (\langle A^+, A^- \rangle \top \langle C^+, C^- \rangle)$

(Def. of $\perp$)

$\equiv \langle A^+ \cup (B^+ \cap C^+), A^- \cap (B^- \cup C^-) \rangle = \langle A^+ \cup B^+, A^- \cap B^- \rangle \perp \langle A^+ \cup C^+, A^- \cap C^- \rangle$

(Def. of $\top$)

$\equiv \langle A^+ \cup (B^+ \cap C^+), A^- \cap (B^- \cup C^-) \rangle =$

$\langle (A^+ \cup B^+) \cap (A^+ \cup C^+), (A^- \cap B^-) \cup (A^- \cap C^-) \rangle$ **(Def. of $\perp$)**

$\equiv \langle (A^+ \cup B^+) \cap (A^+ \cup C^+), (A^- \cap B^-) \cup (A^- \cap C^-) \rangle =$

$\langle (A^+ \cup B^+) \cap (A^+ \cup C^+), (A^- \cap B^-) \cup (A^- \cap C^-) \rangle$ **(Distribute)**

□

32. $\perp$ distributes over $\top$: $A \perp (B \top C) = (A \perp B) \top (A \perp C)$

Proof: $A \perp (B \top C) = (A \perp B) \top (A \perp C)$

$\equiv \langle A^+, A^- \rangle \perp \langle B^+ \cup C^+, B^- \cap C^- \rangle = (\langle A^+, A^- \rangle \perp \langle B^+, B^- \rangle) \top (\langle A^+, A^- \rangle \perp \langle C^+, C^- \rangle)$

(Def. of $\top$)

$\equiv \langle A^+ \cap (B^+ \cup C^+), A^- \cup (B^- \cap C^-) \rangle = \langle A^+ \cap B^+, A^- \cup B^- \rangle \top \langle A^+ \cap C^+, A^- \cup C^- \rangle$

(Def. of $\perp$)

$$\equiv \langle A^+ \cap (B^+ \cup C^+), A^- \cup (B^- \cap C^-) \rangle = \langle (A^+ \cap B^+) \cup (A^+ \cap C^+), (A^- \cup B^-) \cap (A^- \cup C^-) \rangle$$

(Def. of $\top$)

$$\equiv \langle (A^+ \cap B^+) \cup (A^+ \cap C^+), (A^- \cup B^-) \cap (A^- \cup C^-) \rangle =$$

$$\langle (A^+ \cap B^+) \cup (A^+ \cap C^+), (A^- \cup B^-) \cap (A^- \cup C^-) \rangle \qquad \textbf{(Distribute)}$$

□

Theorem 3.1 and Theorem 4.2 assist in characterizing the signed regular languages and operations in terms of abstract algebra.

Definition 4.5 (*Commutative Semigroup [14]*). A pair $\langle G, \circ \rangle$, where G is a set and $\circ$ is a binary operation, is a commutative semigroup if all of the following hold:

1. Closure: $\forall a, b \in G : (a \circ b) \in G$.
2. Associativity: $\forall a, b, c \in G : a \circ (b \circ c) = (a \circ b) \circ c$.
3. Commutativity: $\forall a, b \in G : a \circ b = b \circ a$.

Note that $\langle G, \circ \rangle$ satisfying all of the above except commutativity is called a "semigroup".

Definition 4.6 (*Commutative Monoid [14]*). A triple $\langle G, \circ, e_g \rangle$, where $e_g \in G$ is a commutative monoid if all of the following hold:

1. $\langle G, \circ \rangle$ is a commutative semigroup.
2. Identity: $\forall a \in G : a \circ e_g = a$

That is, (commutative) monoids simply extend (commutative) semigroups with an identity element. We now prove that the four basic binary signed regular operations are commutative monoids with respect to the set of all signed regular languages.

Theorem 4.3 ($\langle \mathfrak{R}^{\pm}, \pm, \theta \rangle$ *is a commutative monoid*).

Proof: $\langle \mathfrak{R}^{\pm}, \pm, \theta \rangle$ satisfies Definition 4.6.

1. $\mathfrak{R}^{\pm}$ is closed under $\pm$. (By Theorem 3.1)
2. $\pm$ is associative for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.5)

3. $\pm$ is commutative for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.1)

4. θ is an identity element of $\pm$ on $\mathfrak{R}^{\pm}$. (By Theorem 4.2.16)

□

Theorem 4.4 ($\langle \mathfrak{R}^{\pm}, \mp, \theta \rangle$ *is a commutative monoid*).

Proof: $\langle \mathfrak{R}^{\pm}, \mp, \theta \rangle$ satisfies Definition 4.6.

1. $\mathfrak{R}^{\pm}$ is closed under $\mp$. (By Theorem 3.1)

2. $\mp$ is associative for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.6)

3. $\mp$ is commutative for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.2)

4. θ is an identity element of $\mp$ on $\mathfrak{R}^{\pm}$. (By Theorem 4.2.15)

□

Theorem 4.5 ($\langle \mathfrak{R}^{\pm}, \top, \langle \emptyset, \Sigma^{+} \rangle \rangle$ *is a commutative monoid*).

Proof: $\langle \mathfrak{R}^{\pm}, \top, \langle \emptyset, \Sigma^{+} \rangle \rangle$ satisfies Definition 4.6.

1. $\mathfrak{R}^{\pm}$ is closed under $\top$. (By Theorem 3.1)

2. $\top$ is associative for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.8)

3. $\top$ is commutative for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.4)

4. $\langle \emptyset, \Sigma^{+} \rangle$ is an identity element of $\top$ on $\mathfrak{R}^{\pm}$. (By Theorem 4.2.18)

□

Theorem 4.6 ($\langle \mathfrak{R}^{\pm}, \perp, \langle \Sigma^{+}, \emptyset \rangle \rangle$ *is a commutative monoid*).

Proof: $\langle \mathfrak{R}^{\pm}, \perp, \langle \emptyset, \Sigma^{+} \rangle \rangle$ satisfies Definition 4.6.

1. $\mathfrak{R}^{\pm}$ is closed under $\perp$. (By Theorem 3.1)

2. $\perp$ is associative for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.7)

3. $\perp$ is commutative for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.3)

4. $\langle \Sigma^+, \emptyset \rangle$ is an identity element of $\perp$ on $\mathfrak{R}^{\pm}$. (By Theorem 4.2.17)

□

Definition 4.7 (*Rig [15]*). A triple $\langle G, +, * \rangle$ is a rig if all of the following hold:

1. $\langle G, +, 0 \rangle$ is a commutative monoid.
2. $\langle G, * \rangle$ is a semigroup
3. Distributivity of $*$ over $+$: $\forall a, b, c \in G : a * (b + c) = (a * b) + (a * c) = (b + c) * a$.

* Note that this definition of *rig* is derived by simply dropping the negativity condition ($\forall a \in G : \exists a^{-1} \in G : a + a^{-1} = 0$) from the definition of a *ring*. Some definitions of rigs insist that 0 is an absorbing element for $*$ (i.e., $\forall a \in G : a * 0 = 0 * a = 0$), but [15] reserves the term *hemiring* for such a structure.

Theorem 4.7 ($\langle \mathfrak{R}^{\pm}, \mp, \perp, \rangle$ *is a rig*).

Proof: $\langle \mathfrak{R}^{\pm}, \mp, \perp, \rangle$ satisfies Definition 4.7.

1. $\langle \mathfrak{R}^{\pm}, \mp, \theta \rangle$ is a commutative monoid. (By Theorem 4.4)
2. $\langle \mathfrak{R}^{\pm}, \perp \rangle$ is a semigroup. (Subsumed by Theorem 4.6)
3. $\perp$ distributes over $\mp$ for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.25 and Theorem 4.2.3)

* Observe that $\langle \mathfrak{R}^{\pm}, \perp, \mp, \rangle$ is also a rig. The commutative monoid and semigroup conditions are still satisfied by the given theorems and $\mp$ distributes over $\perp$ by Theorem 4.2.26 and Theorem 4.2.2. □

Theorem 4.8 ($\langle \mathfrak{R}^{\pm}, \pm, \top, \rangle$ *is a rig*).

Proof:$\langle \mathfrak{R}^{\pm}, \pm, \top, \rangle$ satisfies Definition 4.7.

1. $\langle \mathfrak{R}^{\pm}, \pm, \theta \rangle$ is a commutative semigroup. (Subsumed by Theorem 4.3)
2. $\langle \mathfrak{R}^{\pm}, \top \rangle$ is a semigroup. (Subsumed by Theorem 4.5)
3. $\perp$ distributes over $\mp$ for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.27 and Theorem 4.2.4)

* Observe that $\langle \mathfrak{R}^{\pm}, \top, \pm, \rangle$ is also a rig. The commutative monoid and semigroup conditions are still satisfied by the given theorems and $\pm$ distributes over $\top$ by Theorem 4.2.28 and Theorem 4.2.1. □

Definition 4.8 (*Semilattice [32]*). A pair $\langle G, \circ \rangle$ is a semilattice if all of the following hold:

1. $\langle G, \circ \rangle$ is a commutative semigroup.
2. Idempotence: $\forall a \in G : a \circ a = a$

Theorem 4.9 ($\langle \mathfrak{R}^{\pm}, \pm, \rangle$ *is a semilattice*).

Proof:$\langle \mathfrak{R}^{\pm}, \pm, \rangle$ satisfies Definition 4.8.

1. $\langle \mathfrak{R}^{\pm}, \pm \rangle$ is a commutative semigroup. (Subsumed by Theorem 4.3)
2. $\pm$ is idempotent for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.21)

□

Theorem 4.10 ($\langle \mathfrak{R}^{\pm}, \mp, \rangle$ *is a semilattice*).

Proof:$\langle \mathfrak{R}^{\pm}, \mp, \rangle$ satisfies Definition 4.8.

1. $\langle \mathfrak{R}^{\pm}, \mp \rangle$ is a commutative semigroup. (Subsumed by Theorem 4.4)
2. $\mp$ is idempotent for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.22)

□

Theorem 4.11 ($\langle \mathfrak{R}^{\pm}, \top, \rangle$ *is a semilattice*).

Proof:$\langle \mathfrak{R}^{\pm}, \top, \rangle$ satisfies Definition 4.8.

1. $\langle \mathfrak{R}^{\pm}, \top \rangle$ is a commutative semigroup. (Subsumed by Theorem 4.5)
2. $\top$ is idempotent for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.23)

□

Theorem 4.12 ($\langle \mathfrak{R}^{\pm}, \bot, \rangle$ *is a semilattice*).

Proof:$\langle \mathfrak{R}^{\pm}, \bot, \rangle$ satisfies Definition 4.8.

1. $\langle \mathfrak{R}^{\pm}, \top \rangle$ is a commutative semigroup. (Subsumed by Theorem 4.6)

2. $\top$ is idempotent for elements of $\mathfrak{R}^{\pm}$. (By Theorem 4.2.24)

□

Definition 4.9 (*Bounded Distributive Lattice [16]*). A triple $\langle G, \vee, \wedge \rangle$ is a bounded distributive lattice if all of the following hold:

1. $\langle G, \vee \rangle$ is a semilattice.

2. $\langle G, \wedge \rangle$ is a semilattice.

3. Absorption Law 1: $\forall a, b \in G : a \vee (a \wedge b) = a$

4. Absorption Law 2: $\forall a, b \in G : a \wedge (a \vee b) = a$

5. Identity for $\vee$: $\exists 0 \in G : \forall a \in G : a \vee 0 = a$

6. Identity for $\wedge$: $\exists 1 \in G : \forall a \in G : a \wedge 1 = a$

   * The upper and lower bounds of the lattice are then 1 and 0, respectively.

7. Distributivity of $\vee$ over $\wedge$: $\forall a, b, c \in G : a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$.

8. Distributivity of $\wedge$ over $\vee$: $\forall a, b, c \in G : a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.

Theorem 4.13 ($\langle \mathfrak{R}^{\pm}, \top, \bot, \rangle$ *is a bounded distributive lattice*).

Proof:$\langle \mathfrak{R}^{\pm}, \top, \bot, \rangle$ satisfies Definition 4.9.

1. $\langle G, \top \rangle$ is a semilattice. (By Theorem 4.11)

2. $\langle G, \bot \rangle$ is a semilattice. (By Theorem 4.12)

3. $\forall a, b \in G : a \top (a \bot b) = a$. (By Theorem 4.2.29)

4. $\forall a, b \in G : a \bot (a \top b) = a$. (By Theorem 4.2.30)

5. $\langle \emptyset, \Sigma^{+} \rangle$ is an identity for $\top$. (By Theorem 4.2.18)

6. $\langle \Sigma^+, \emptyset \rangle$ is an identity for $\perp$. (By Theorem 4.2.17)

7. $\top$ distributes over $\perp$. (By Theorem 4.2.31)

8. $\perp$ distributes over $\top$. (By Theorem 4.2.32)

□

Now that we have identified some algebraic structures formed by the signed regular languages under various operations, we can leverage results pertaining to these structures in order to better understand them. For example, it is shown in [40] that any monoid $\langle G, \circ, e_g \rangle$ can be augmented such that $\circ$ operates on subsets of G as follows:

$$\forall X, Y \subseteq G : X \circ Y = \{x \circ y | x \in X, y \in Y\}$$

That is, $\langle \mathcal{P}(G), \circ, \{e_g\} \rangle$ is also a monoid; an analogous result holds for semigroups. What this means to us is that all four of our basic binary operations can be easily extended to operate on collections of signed regular languages. This generalization may be useful if we wanted combinators with a variable number of subpolicies (e.g., a list of them).

It is difficult to imagine the exact shape of the lattice $\langle \mathfrak{R}^{\pm}, \top, \perp \rangle$ because it is infinite. Because this lattice is distributive, we know that the two shapes depicted in Figure 4.3, called a pentagon and a diamond, do not appear as sub-structures anywhere in the lattice [16]. Figure 4.4 demonstrates the shape of the lattice if $\mathfrak{R}^{\pm}$ was the finite set $\{a, b, c\}$ to assist in understanding its overall form. Additionally, we obtain (from [16]) two equivalent formulations of the relation $\leq$ on arbitrary $a, b \in \mathfrak{R}^{\pm}$ that we can use to order the signed regular languages:

$$a \leq b \Leftrightarrow a \perp b = a$$
$$a \leq b \Leftrightarrow a \top b = b$$

We expected that $\langle \mathfrak{R}^{\pm}, \pm, \mp \rangle$ would form some type of interesting structure because the union-like operations mirror each other in the venn-diagrams just as $\top$ and $\perp$ do; they even satisfy De Morgan's laws with $\neg$. However, neither union-like operation distributes over the other, so they fail to form even a *ringoid*—a minimal ring-like structure in which one binary

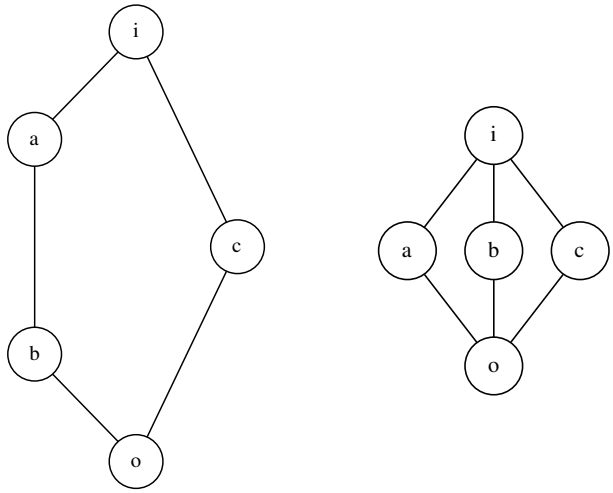

Figure 4.3. Pentagon and diamond shapes, which are not sub-lattices of $\langle \mathfrak{R}^{\pm}, \top, \bot \rangle$. The variables a, b, c, i, and o are arbitrary signed regular languages. This figure is a re-creation of Figure 2 in [16]

.

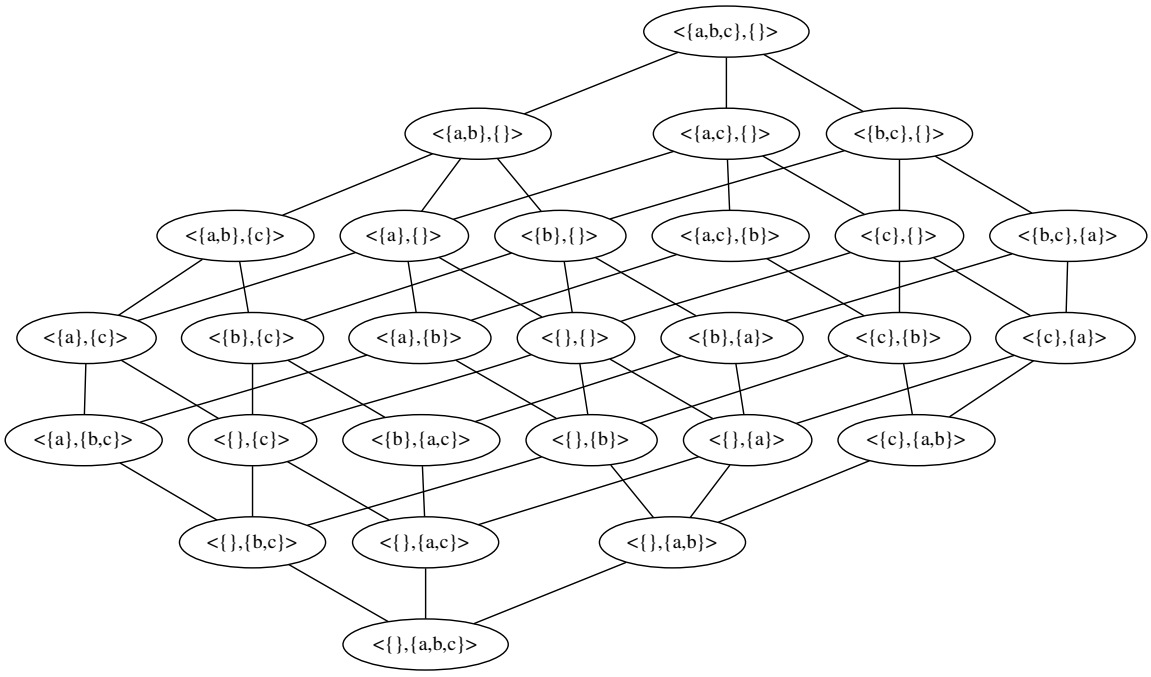

Figure 4.4. Lattice for $\langle \{a, b, c\}, \top, \bot \rangle$.

operation distributes over another. An intersection operation that combines signed regular languages $\langle A^+, A^- \rangle$ and $\langle B^+, B^- \rangle$ into $\langle A^+ \cap B^+, A^- \cap B^- \rangle$ would form hemirings (see Definition 4.7) with both $\pm$ and $\mp$, individually, but such an operation is special purpose and non-essential. Although $\langle \mathfrak{R}^{\pm}, \pm \rangle$ and $\langle \mathfrak{R}^{\pm}, \mp \rangle$ form semilattices, $\langle \mathfrak{R}^{\pm}, \pm, \mp \rangle$ fails the absorption laws and is not a lattice. Each of the union-like operations does form a rig with either $\bot$ or $\top$ and they are also useful for composition. Their utility is evident from their frequency in the PoCo policies (Chapter 5).

It is satisfying that conjunction and disjunction form a bounded distributive lattice because it is a rich structure (i.e., it has many conditions), and they embody the two natural strategies for a security language to combinine two policies. Similarly, given that inversion is the most common unary operation on policies, we are glad that it is involutary and distributes over each of the binary operations (i.e., De Morgan's laws hold). Now that we have studied the properties of these operations, we consider their capabilities in the next section.

4.5 The Functionally Complete Operations For Signed Regular Languages

We continue the analysis of operations on signed regular languages in this section by considering their expressive power. That is, we would like to know whether these operations are functionally complete—sufficient for expressing all of the valid functions on the component sets of signed regular languages. Moreover, we are interested in determining the minimal, functionally complete operation set(s).

We can think of each signed regular language as a function $f : E \to \{+, 0, -\}$ and any n-ary operation on them as a function $g : \{+, 0, -\}^n \to \{+, 0, -\}$. That is, signed regular languages place each event into one of three categories: positive, neutral, or negative; operations on these languages combine their categorizations to form new ones. We are now poised to analyze signed regular operations as functions in a three-valued logic.

Some useful techniques from Boolean logic carry over if we consider these signs as truth values, where $+$ is $true$, $-$ is $false$, and 0 is an intermediary value between them. Opera-

tions can then be described by truth tables, where signed regular languages are represented as variables over $\{+, 0, -\}$. These truth tables can be decomposed into formulas in disjunctive normal form (DNF), in a manner similar to Boolean operations, using the $\bot$ and $\top$ operations along with a new $=$ operation. That is to say, $\{\bot, \top, =\}$ is functionally complete. The following truth table describes these operations in our three-valued logic:

p	q	$p = q$	$p \top q$	$p \bot q$
$-$	$-$	$+$	$-$	$-$
$-$	0	$-$	0	$-$
$-$	$+$	$-$	$+$	$-$
0	$-$	$-$	0	$-$
0	0	$+$	0	0
0	$+$	$-$	$+$	0
$+$	$-$	$-$	$+$	$-$
$+$	0	$-$	$+$	0
$+$	$+$	$+$	$+$	$+$

The following facts about the equality, disjunction, and conjunction operations make them suitable for representing any truth table:

1. $=$ has co-domain $\{+, -\}$.
2. $- \top x \equiv x \top - \equiv x$
3. $+ \top x \equiv x \top + \equiv +$.
4. $- \bot x \equiv -$
5. $+ \bot x \equiv x$.

The constructions that follow rely on the constant functions f^+, f^0, and f^- representing the signed regular languages $\langle \Sigma^+, \emptyset \rangle$, θ, and $\langle \emptyset, \Sigma^+ \rangle$, respectively. These constants are trivial to implement in PoCo, so their availability is assured rather than assumed. Now, we can represent arbitrary truth tables as formulae in disjunctive normal form. For example, the

following formula, where each disjunct is aligned with its corresponding row in the adjacent truth table, is equivalent to $p \pm q$:

p	q	$p \pm q$	
$-$	$-$	$-$	$((p = f^-)\bot(q = f^-)\bot f^-)\top$
$-$	0	$-$	$((p = f^-)\bot(q = f^0)\bot f^-)\top$
$-$	$+$	$+$	$((p = f^-)\bot(q = f^+)\bot f^+)\top$
0	$-$	$-$	$((p = f^0)\bot(q = f^-)\bot f^-)\top$
0	0	0	$((p = f^0)\bot(q = f^0)\bot f^0)\top$
0	$+$	$+$	$((p = f^0)\bot(q = f^+)\bot f^+)\top$
$+$	$-$	$+$	$((p = f^+)\bot(q = f^-)\bot f^+)\top$
$+$	0	$+$	$((p = f^+)\bot(q = f^0)\bot f^+)\top$
$+$	$+$	$+$	$((p = f^+)\bot(q = f^+)\bot f^+)$

The disjuncts, like the truth table, permute all possible values of p and q and test them against constants. Therefore, exactly one of the disjuncts in the formula will satisfy (i.e., evaluate to $+$ on) all of the equality tests and attain the form $(+\bot + \bot x)$, which is equivalent to x. All other disjuncts will fail (i.e., evaluate to $-$ on) at least one equality test and evaluate to $(-\bot a\bot y)$, $(a\bot - \bot y)$, or $(-\bot - \bot y)$, which is equivalent to $-$ in any case. The overall formula then reduces to $(-\top - \cdots \top x\top \cdots - \top -)$, which is equivalent to x, the desired value.

The set $\{\top, \bot, =\}$ is functionally complete because this strategy generalizes for any n-ary operation. The following truth table and formula demonstrate the construction in the abstract:

p_1	p_2	$\cdots$	p_n	$f(p_1, p_2, \ldots, p_n)$	
$-$	$-$	$\cdots$	$-$	v	$((p_1 = f^-)\bot(p_2 = f^-)\bot\cdots\bot(p_n = f^-)\bot v)\top$
$-$	$-$	$\cdots$	0	w	$((p_1 = f^-)\bot(p_2 = f^-)\bot\cdots\bot(p_n = f^0)\bot w)\top$
$-$	$-$	$\cdots$	$+$	x	$((p_1 = f^-)\bot(p_2 = f^-)\bot\cdots\bot(p_n = f^+)\bot x)\top$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
0	0	$\cdots$	0	y	$((p_1 = f^0)\bot(p_2 = f^0)\bot\cdots\bot(p_n = f^0)\bot y)\top$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$+$	$+$	$\cdots$	$+$	z	$((p_1 = f^+)\bot(p_2 = f^+)\bot\cdots\bot(p_n = f^+)\bot z)$

The question of whether our original signed regular operations are functionally complete can now be answered. Recall that our DNF construction relies on an equality operation, which was introduced just for that purpose. Table 4.1 depicts a truth table demonstrating the equivalence of $p = q$ and $((p\bot q) \pm ((\neg p)\bot(\neg q))) \mp f^+$. Therefore, the DNF construction can be performed using only our original operations.

Table 4.1. Truth table for $p = q$ expressed in terms of our original signed regular operations.

p	q	$((p$	$\bot$	$q)$	$\pm$	$((\neg$	$p)$	$\bot$	$(\neg$	$q)))$	$\mp$	f^+
$-$	$-$	$-$	$-$	$-$	$+$	$+$	$-$	$+$	$+$	$-$	$+$	$+$
$-$	0	$-$	$-$	0	$-$	$+$	$-$	0	0	0	$-$	$+$
$-$	$+$	$-$	$-$	$+$	$-$	$+$	$-$	$-$	$-$	$+$	$-$	$+$
0	$-$	0	$-$	$-$	$-$	0	0	0	$+$	$-$	$-$	$+$
0	0	0	0	0	0	0	0	0	0	0	$+$	$+$
0	$+$	0	0	$+$	$-$	0	0	$-$	$-$	$+$	$-$	$+$
$+$	$-$	$+$	$-$	$-$	$-$	$-$	$+$	$-$	$+$	$-$	$-$	$+$
$+$	0	$+$	0	0	$-$	$-$	$+$	$-$	0	0	$-$	$+$
$+$	$+$	$+$	$+$	$+$	$+$	$-$	$+$	$-$	$-$	$+$	$+$	$+$

Now that it has been established that $\{\mp, \pm, \top, \bot, \neg\}$ is a functionally complete set of operations, we are in a position to consider a minimal set from which each of them can

be constructed. In fact, there is a functionally complete singleton (often called a "Sheffer function" due to [37]) $\{\star\}$, where $\star$ is defined by the following truth table:

p	q	$p \star q$
$-$	$-$	$+$
$-$	0	0
$-$	$+$	$-$
0	$-$	$-$
0	0	$+$
0	$+$	0
$+$	$-$	$-$
$+$	0	0
$+$	$+$	$+$

Before the $\star$ constructions for conjunction, disjunction, and equality are presented, notice that we only need them to be equivalent in the cases that occur in the DNF formulae. That is, the equivalence for $\bot$ doesn't need to hold when the left operand is 0 because its left-hand side in such a formula is always the outcome of an equality test. Similarly, one of $\top$'s operands will certainly be $-$, so other permutations aren't necessary to consider. Ignoring these impossible permutations reduces the complexity of the equivalent constructions.

The following equivalences hold for $=$, $\bot$, and $\top$ as they are used in the DNF formulae (Tables 4.2, 4.3, and 4.4 depict their respective truth tables):

1. $p = q \equiv (p \star q) \star (f^- \star ((f^- \star (p \star q)) \star (p \star q)$
2. $p \bot q \equiv (p \star ((f^- \star (f^0 \star (f^- \star p))) \star (q \star p))) \star p$
3. $p \top q \equiv f^- \star ((f^- \star p) \star (f^- \star q))$

We now know that we could implement all operations with just $\star$. This fact gives us some peace of mind in that we don't have to worry about useful combinators that were overlooked in the design process. Additionally, an implementation of PoCo could provide

Table 4.2. Truth table for $p = q$ expressed in terms of $\star$.

p	q	$(p$	$\star$	$q)$	$\star$	$(f^-$	$\star$	$((f^-$	$\star$	$(p$	$\star$	$q))$	$\star$	$(p$	$\star$	$q)))$
$-$	$-$	$-$	$+$	$-$	$+$	$-$	$+$	$-$	$-$	$-$	$+$	$-$	$-$	$-$	$+$	$-$
$-$	0	$-$	0	0	$-$	$-$	$-$	$-$	0	$-$	0	0	$+$	$-$	0	0
$-$	$+$	$-$	$-$	$+$	$-$	$-$	$+$	$-$	$+$	$-$	$-$	$+$	$-$	$-$	$-$	$+$
0	$-$	0	$-$	$-$	$-$	$-$	$+$	$-$	$+$	0	$-$	$-$	$-$	0	$-$	$-$
0	0	0	$+$	0	$+$	$-$	$+$	$-$	$-$	0	$+$	0	$-$	0	$+$	0
0	$+$	0	0	$+$	$-$	$-$	$-$	$-$	0	0	0	$+$	$+$	0	0	$+$
$+$	$-$	$+$	$-$	$-$	$-$	$-$	$+$	$-$	$+$	$+$	$-$	$-$	$-$	$+$	$-$	$-$
$+$	0	$+$	0	0	$-$	$-$	$-$	$-$	0	$+$	0	0	$+$	$+$	0	0
$+$	$+$	$+$	$+$	$+$	$+$	$-$	$+$	$-$	$-$	$+$	$+$	$+$	$-$	$+$	$+$	$+$

Table 4.3. Truth table for $p \perp q$ expressed in terms of $\star$.

p	q	$(p$	$\star$	$((f^-$	$\star$	$(f^0$	$\star$	$(f^-$	$\star$	$p)))$	$\star$	$(q$	$\star$	$p)))$	$\star$	p
$-$	$-$	$-$	0	$-$	0	0	0	$-$	$+$	$-$	0	$-$	$+$	$-$	$-$	$-$
$-$	0	$-$	$+$	$-$	0	0	0	$-$	$+$	$-$	$-$	0	$-$	$-$	$-$	$-$
$-$	$+$	$-$	$+$	$-$	0	0	0	$-$	$+$	$-$	$-$	$+$	$-$	$-$	$-$	$-$
$+$	$-$	$+$	$-$	$-$	$+$	0	$-$	$-$	$-$	$+$	$-$	$-$	$-$	$+$	$-$	$+$
$+$	0	$+$	0	$-$	$+$	0	$-$	$-$	$-$	$+$	0	0	0	$+$	0	$+$
$+$	$+$	$+$	$+$	$-$	$+$	0	$-$	$-$	$-$	$+$	$+$	$+$	$+$	$+$	$+$	$+$

Table 4.4. Truth table for $p \top q$ expressed in terms of $\star$.

p	q	f^-	$\star$	$((f^-$	$\star$	$p)$	$\star$	$(f^-$	$\star$	$q))$
$-$	$-$	$-$	$-$	$-$	$+$	$-$	$+$	$-$	$+$	$-$
$-$	0	$-$	0	$-$	$+$	$-$	0	$-$	0	0
$-$	$+$	$-$	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$+$
0	$-$	$-$	0	$-$	0	0	0	$-$	$+$	$-$
$+$	$-$	$-$	$+$	$-$	$-$	$+$	$-$	$-$	$+$	$-$

just one combinator without losing any expressiveness. While this is not something to be recommended, as many compositions would become unwieldy, it is still an advantage over other systems. For example, although Polymer provides several general-purpose combinators, we introduced a `Not` combinator in Section 3.3 to make up for the lack of unary inversion (in order to avoid writing two versions of the `OpenWithExt` policy).

Now that sufficient background has been established and our policy outputs have been thoroughly examined, we are ready to put them to use in actual policies. The next chapter defines and explains the PoCo language and demonstrates its features through example policies.

CHAPTER 5

THE POCO LANGUAGE WITH EXAMPLES

This chapter introduces the PoCo language by specifying its syntax and providing examples that demonstrate how its features operate. We begin with an overview of the language, which provides necessary background and walks the reader through the construction of some policies and combinators. The overview demonstrates the use of many of PoCo's features and explains how they operate. Afterward, the full syntax is defined, which includes features that do not appear in the overview. In order to demonstrate its expressive power, we translate a sophisticated policy that constrains the behavior of an email client (from [3]) from Polymer into PoCo.

5.1 Overview of PoCo

Before introducing PoCo's complete syntax, we first discuss the basics of specifying policies in the language. Most of the syntax can be extrapolated from a few examples. We start with a very simple policy to cover the bare essentials of how policies are specified and enforced. Afterward, we relax this policy, which increases its complexity and introduces some new features. We then move on to the specification and application of some example combinators.

PoCo is defined in this dissertation with object-oriented, Java-like applications in mind because Polymer's implementation constrains Java programs. Continuing in the same tradition allows the most direct comparison between the languages. Additionally, AspectJ is appropriate for implementing PoCo, so it is reasonable to proceed accordingly. Therefore, actions have the form of method invocations, and results have the form of objects (which

may be wrappers for primitive values). The specification does not depend on object orientation and could have been presented in terms of another paradigm (e.g., a functional language), but doing so would obscure PoCo's relationship to Polymer.

Recall that an MRA execution is represented as a sequence of exchanges, each consisting of an input and output event. PoCo policies are best thought of as patterns that describe all satisfactory executions. Policies are expressed using *abstract exchanges*, each of which is a pair containing a regular language and a signed regular language (in that order). Regular expressions are denoted by ` ... `, and they are signed by a preceding + or −. Signed regular expressions can be combined using the signed regular operations introduced in Section 4.3. Abstract exchanges can be sequenced (specified one after the another), switched on (using |), repeated zero or more (including infinitely-many) times (using superscript ∞), and grouped (using [and]). Each of these constructions is later demonstrated with concrete examples.

The first element in each abstract exchange is a regular expression on the input event and the previous output event, which may be complemented by preceding it with $\sim$. Strings of the form $a \Rightarrow r$ denote that the input r is the result of executing the previously output action a, and those of the form $a \Leftarrow r$ denote that the input action a was received in response to result r being output. This information allows each policy to determine the actual execution that is occurring, which, due to how it is composed with others, may differ from that which it is attempting to build.

The abstract exchanges can include "bindings" of variable names to regular expressions in order to store information from events in memory. These bindings have the following form:

$$@name[regular\ expresson].$$

The values that these variables are bound to can be inserted into (signed) regular expressions using $\$name$. This allows policies to express relationships between various points in executions. Binding and inserting values does not violate the regularity of the languages because insertions happen before the match is attempted and binding occurs only after a

match succeeds. This is in contrast to the "regex" facilities of many modern programming languages that support "back-matching", which is a non-regular behavior.

The policy output, along with its relationship to the input event, indicates how an MRA should operate in order to enforce the policy. Informally, the algorithm for choosing an output event has seven steps:

1. If there is an easy-to-find action that is good, output it.
2. If the input event is good, output it.
3. If there is an easy-to-find result that is good, output it.
4. If the input event is neutral, output it.
5. If there is an easy-to-find result that is neutral, output it.
6. If there is an easy-to-find action that is neutral, output it.
7. Terminate the execution.

The term "easy-to-find" in the above algorithm means that the event is a member of a finite set in the output. Because a signed regular language can have infinite-many positive members and non-members (negative members are never output), it may be the case that we have infinitely-many events to choose from when selecting one to output. We avoid outputting arbitrary members of infinite sets in all cases because they are too liberally defined for each member to be considered a reasonable choice.

We now specify the output-selection algorithm more precisely. Recall that A denotes the set of all actions, R denotes the set of all results, and $E = A \cup R$. Suppose a policy outputs the signed regular language $\langle B^+, B^- \rangle$ in response to the input event e_{in}. Then the monitor proceeds considering each of the following cases in order:

1. Case $(B^+ \cap A)$ non-empty and finite: Output some action $a \in ((B^+ \cap A) \setminus \{e_{in}\})$. Priority is given to actions that the policy proposes, so long as it proposes finitely

many, because policies are essentially deferring their decision on the input action while they output actions of their own and receive results for them.

2. Case $e_{in} \in B^+$: Output e_{in}. The input is preferred by the policy and there is no finite set of proposed alternative actions. Even if e_{in} is a result and some results are in B^+, we prefer e_{in} because only one result can be given back to the application for its action and e_{in} was proposed by both the system and the policy.

3. Case $(B^+ \cap R)$ non-empty and finite: Output some result $r \in (B^+ \cap R)$. When the policy wants to output a result, it is done performing any actions. Any positively signed result is suitable at this point.

4. Case $e_{in} \notin B^-$: Output e_{in}. There is no obvious preferred event to output, so default to the input because it is neutral.

5. Case $(\overline{(B^+ \cup B^-)} \cap R)$ non-empty and finite: Output some result $r \not\in^{\pm} B$. A finite, non-empty set of neutral events will only exist if some particular events were left out of a signed regular expression. Because they are explicitly not signed, they can be anticipated, and are therefore reasonable to output. Neutral results are preferred over neutral actions because there is more incentive to get another action from the target than there is to extend the execution with irrelevant actions.

6. Case $(\overline{(B^+ \cup B^-)} \cap A)$ non-empty and finite: Output some action $a \not\in^{\pm} B$. Because all other outputs have been ruled out by this stage, neutral actions are the only valid outputs to choose from.

7. Otherwise: Output nothing. There is no acceptable course of action, so the monitor must refrain from outputting any event. This effectively terminates an MRA execution because no further events will be input or output. In practice, the monitor could output some abortive action, if one exists.

We have now established the high-level structure of policies and the details of how policy outputs are interpreted by an MRA. We proceed to study concrete policy examples to illustrate how they operate and demonstrate some of PoCo's syntax.

5.1.1 Example Base Policies

A simple class of PoCo policies prohibit certain events from ever being output from the monitor; these can be specified using a single abstract exchange. For example, suppose we wanted to prevent a program from reading the disk as a precaution against searches for sensitive information. Say this occurs by calling the `read()` method of a `File` object. Then the event of interest would be an action matching the pattern $`\{\%\} : File \rightarrow read()`$, where:

1. Virtual method calls have the form $o \rightarrow m$, where m is the method being invoked on some object o.

2. Objects have the form $\{\dots\} : \tau$, where { and } delimit a sequence of field-value associations and τ is some object type.

3. Methods have the form $x(o_1, \dots, o_n)$, where x is its name and each o_i is an argument object.

4. The symbol % is the multi-character wildcard, which matches any (possibly empty) sequence of symbols.

Therefore, the above pattern matches an invocation of `read()` on any instance of `File`, irrespective of the object's field values.

We can specify the policy that prevents the target from reading files as follows:

$$NoFileReads() : (\sim `\{\%\} : File \rightarrow read()`, \theta)^{\infty}$$

This policy is introduced with the name `NoFileReads` and has no parameters. It matches input events against the pattern $\sim `\{\%\} : File \rightarrow read()`$, which successfully matches any input except an invocation of `read()` on a `File`. Upon matching an input, the policy

outputs θ, indicating indifference toward the input. This behavior is repeated as long as more inputs arrive, perhaps infinitely many times. If a file-reading action was input, then the match would fail and the poilcy would cease to output anything; the execution would not progress further.

This policy is likely to be overly restrictive given that there are many legitimate reasons for a program to read a file. The application may read some configuration files upon starting. If so, `NoFileReads` will prohibit this and prevent the program form progressing any further. We can write a more accommodating version called `LimitFileReads` that is parameterized by an integer denoting the maximum number of `File→read()` calls that are permitted.

The overall logic of `LimitFileReads` is illustrated in Figure 5.1. It starts by "waiting" for the application to input `File→read()`, meaning that it outputs θ while other events are input. When `File→read()` is input, the policy determines whether any more file-reading calls are permitted. If so, the file-reading action is output, the number of permitted calls is decremented, and the policy repeats from the beginning; otherwise, the policy fails to match and the execution ceases.

The repetition that occurs whenever the policy wants a certain action to be executed is to cooperate with other policies whose proposed alternative events may get output first. The policy attempts to deliver the result of `File→read()` to the application as long as the monitor is outputting actions to the system. That is, when this policy has finished interjecting its bookkeeping sequence after the file is read, other policies may continue to execute actions.

Now that we have designed the policy logic, we move on to implementing it. The realization of `LimitFileReads` introduces several new features. The policy is first presented and then explained in detail. We specify this policy, with line numbers for later reference, as follows:

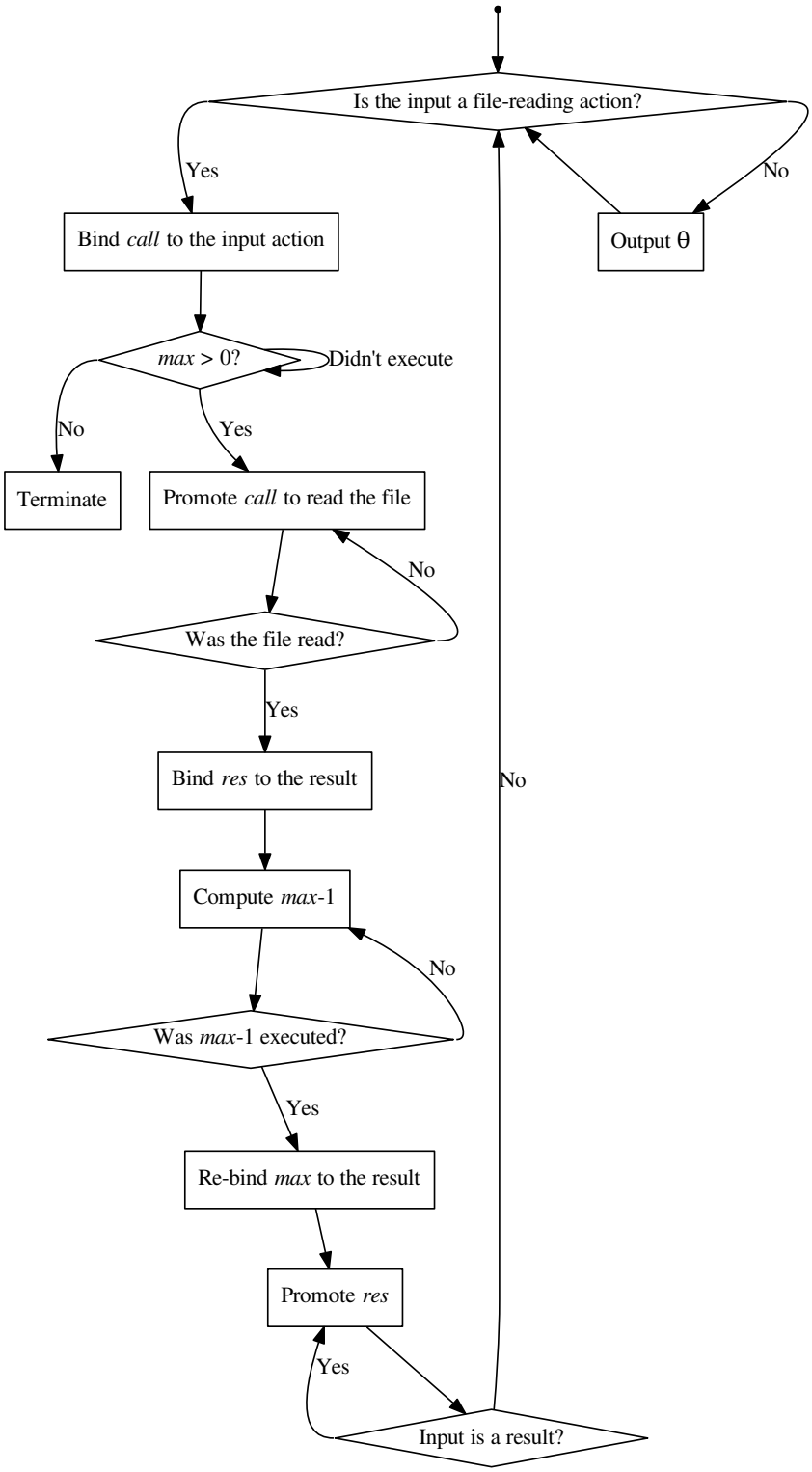

Figure 5.1. An illustration of the logic of the `LimitFileReads` policy.

$LimitFileReads(Integer\ max)$:

1 [(`$@call[\{\%\} : File \rightarrow read() \Leftarrow \%]$`, + + `$gt(\$max, \{0\} : Integer)$`)

2 (`$gt(\$max, \{0\} : Integer) \Rightarrow \{true\} : Boolean$`, + + `$\$call$`)

3 (`$\$call \Rightarrow @res[\%]$`, + + `$sub(\$max, \{1\} : Integer)$`)

4 (`$sub(\$max, \{1\} : Integer) \Rightarrow @max[\%]$`, +`$\$res$`)

5 (`$\% \Rightarrow \%$`, +`$\$res$`)$^{\infty}$

6 | (`$\%$`, θ)]$^{\infty}$

The policy code contains the syntactic construct ++, which has not yet been discussed. Whereas +`...` denotes a positive, signed regular language, + + `...` further indicates that the action therein is required before moving on in the policy. This feature is merely "syntactic sugar": it is not strictly necessary. In fact, (` ... `, + + `a`) is merely shorthand for (` ... `, +`a`)(∼`a ⇒ %`, +`a`)$^{\infty}$. That is, the required action a is output, and it is continually proposed until its result is input (i.e., it was executed). When a is executed, the pattern ∼ `a ⇒ %` will fail to match its result because it is the complement of exactly that. The policy will stop trying to repeat the (∼`a ⇒ %`, +`a`) abstract exchange when the result arrives, and will move on to attempt matching it against the next part of the policy.

Because the only operators that can appear within abstract exchanges are for combining signed regular languages, PoCo lacks expressions for performing general computations and comparing or modifying the values of variables. Computations are performed by outputting the appropriate actions, and variable are updated when identifiers are bound to new values. `LimitFileReads` uses the `gt` and `sub` actions to perform greater-than testing and subtraction, respectively. Although this restriction makes policy specification less convenient, it makes even policy-state computations composable.

We now examine each line of the policy in greater detail, line by line:

1. If the input event is an invocation of a `File` object's `read()` method, then bind it to *call*, promote the action that tests whether max is greater than zero repeatedly until it executes, and go to Line 2; otherwise, go to Line 6.

2. If the greater-than test (which was required on Line 1 and must have executed by now) returns `true` (e.g., $max > 0$), then promote *call* (the input action from Line 1) repeatedly until it executes and go to Line 3; otherwise, the match fails because the inequality test returned `false`. The matching behavior falls through as follows:

   (a) The pattern `` `$greater(\$max, \{0\} : Integer) \Rightarrow \{true\} : Boolean$` `` doesn't match the input when the result is $\{false\} : Boolean$.

   (b) This abstract exchange is not part of a switch. Although Line 1 and Line 6 form a switch, matching on the beginning of the sequence already succeeded on Line 1 at this point, so there are no further cases to consider if matching fails on Line 2.

   (c) This abstract exchange is part of a repeated sequence (Lines 1-6). When a repeated sequence fails to match an input, the matching attempt continues on whatever follows that sequence. In this case, nothing follows the repeated sequence in which the match failure occurred, so execution ceases.

3. We know that *call* executed if we've reached this point because Line 2 repeatedly promotes it in all other cases. Bind the result of *call* to *res* so we can output it later, promote computing $max - 1$ repeatedly until it executes, and go to Line 4.

4. It must be that $max - 1$ was computed at this point, so re-bind max to the result. We are done with our bookkeeping routine now and can give the application the result for the `read()` invocation that occurred earlier when *call* executed. Promote *res* and go to Line 5.

5. While actions are being executed (output by other policies), continue to promote *res*. Go to Line 1 when an action is input (the monitor output a result to the application).

6. The input didn't match `File→read()` on Line 1, so we catch all other possibilities here and output θ because we are only concerned with file reading. Go back to Line 1.

If `LimitFileReads` was concerned with not only the target, but other policies reading files, it would need to prohibit `read()` when outputting each of its bookkeeping actions, and decrement *max* whenever `read()` is executed due to other policies. The way the policy is currently specified, it neither negatively signs `read()` in its outputs nor checks whether an input is the result of a `read()` action. Even when $\$call$ is promoted, all other file reads would need to be prohibited (the $\pm$ operation is useful for making such exceptions). This is because policies do not respond to each other's outputs, but express themselves fully upfront, letting combinators control how those outputs are composed. Therefore, a prohibition-preserving operation like $\mp$ or $\perp$ could be used to combine `LimitFileReads` with another policy in order to control whether that policy, as well as the target, may read files.

This prohibition is not necessary when constraining the application only, because a positively-signed, finite set of output actions takes precedence over any policy input, as previously discussed. That is, when a `read()` action is input on Line 1, the positively-signed `gt` action will be output by the monitor. If the `read()` turns out to be permissible after *max* is checked, then the policy has to propose it because the current input event at that point will be the result of `gt`. No result will be output to the application while the policy outputs positively-signed actions.

5.1.2 Example Combinators

Combinators are defined similarly to policies, but at least one of their parameters is a policy. In addition to matching on the input event, a combinator also matches against the

signed regular outputs of its subpolicies. As a result, the abstract exchanges in combinators are triples, rather than pairs, containing a tuple of subpolicy constraints in between the usual regular input and signed regular output. The subpolicy identifiers can be used as signed regular languages within the triple, where they represent the subpolicy outputs on the current input event. There are three ways to specify a constraint on a subpolicy output:

1. Assignment (e.g., $x \leftarrow s$): Binds an identifier x to a signed regular language s. This constraint is always satisfied, and the bindings last until they are overwritten.

2. Signed membership (e.g., $r \in^{+} s$): The membership of regular language r, with respect to the positive, negative, and neutral components of signed regular language s, can be tested. Our use of the term "membership" here is rather liberal because r may contain any number of events. Therefore, we are checking whether r is a subset of the given component.

3. Signed regular equality (e.g., $x = +` \ldots ` \pm -` \ldots `$): The policy output can be compared in terms of equality (including $\neq$) against another signed regular language.

Alternatively, a wildcard (written $\_$), which is trivially satisfied, may be given in place of the constraint tuple.

For example, consider a combinator called `PreferOKInputAct` with a single subpolicy that prefers the input action instead of its subpolicy's output whenever the input is not prohibited by the subpolicy (i.e., it is "OK"). When the input is a result or is prohibited by the subpolicy, it outputs whatever the subpolicy does. This combinator has three cases to consider:

1. The input is an action, and it is negatively signed in the subpolicy output.

2. The input is an action, and it is not negatively signed in the subpolicy output (i.e., it is positive or neutral).

3. The input is a result.

We are now ready to specify this combinator. We can avoid grouping the entire policy in order to repeat its behavior by indicating repetition after the policy parameters (e.g., $name(parameter, \ldots, parameter)^{\infty} : \ldots$). The final case in the combinator doesn't have to actually specify that the input is a result because the first two cases handle all possible actions. Because the cases are considered in order when matching, the pattern `` `%` `` is sufficient there. We can specify this combinator as follows:

$$
\begin{aligned}
&PreferOKInputAct(Policy\ p)^{\infty} : \\
&\quad (\texttt{`@}in[\%] \Leftarrow \texttt{\%`}, (\texttt{`\$}in\texttt{`} \in^{-} p), p) \\
&\mid (\texttt{`@}in[\%] \Leftarrow \texttt{\%`}, \_, +\texttt{`\$}in\texttt{`}) \\
&\mid (\texttt{`\%`}, \_, p)
\end{aligned}
$$

Combinators output a signed regular language just as policies do. The five standard signed regular operations have natural combinator analogues. For example, we can define a conjunctive combinator as follows:

$$Conj(Policy\ p1, Policy\ p2)^{\infty} : (\texttt{`\%`}, \_, p1 \perp p2)$$

This `Conj` combinator takes two policy parameters and no others. On any input event, it outputs $p1 \perp p2$, which is the conjunction of its subpolicies' outputs on that input. Examples of more sophisticated combinators will appear when we implement superpolicies from Polymer's `Email` policy using PoCo.

Policies may also be composed by the literal abstract exchanges that make them up. That is, combinators may treat policies as data in order to form new policies. The declarative style of PoCo policy definitions makes them well-suited for this type of combination. Because abstract exchanges can be switched, concatenated, and repeated, it is natural to define combinators that combine entire policies in each of these ways. The concatenation combinator, for example, simply follows the first policy body with the second. Being able to combine policies literally allows us to express some complex policies as combinations of single-exchange policies. The three main literal combinators are straightforward:

$$Switch(Policy\ p1, Policy\ p2) : [\ p1 \mid p2\ ]$$

$$Cat(Policy\ p1, Policy\ p2) : p1\ p2$$

$$Rep(Policy\ p) : [p]^{\infty}$$

The usefulness of literal combination can be demonstrated using our `LimitFileReads` policy. Encoding the `read()` limit via the *max* variable led to more complication than the policy warrants. We can instead write a new policy that permits at most one invocation of `read()` and concatenate it with itself to express the number of invocations that are allowed as follows:

$$OneFileRead() :$$
$$(\sim \grave{}\{\%\} : File{\rightarrow}read() \Leftarrow \%\grave{}, \theta)^{\infty}(\grave{}\%\grave{}, \theta)(\sim \grave{}\{\%\} : File{\rightarrow}read() \Leftarrow \%\grave{}, \theta)^{\infty}$$

Now, we can compose `OneFileRead` with itself to limit the application to $k \in \mathbb{N}$ file-reading operations. For example, the application can be limited to reading three files with the following composition:

$$Cat(OneFileRead(), Cat(OneFileRead(), OneFileRead()))$$

The above composition expands to a sequence of abstract exchanges:

$$(\sim \grave{}\{\%\} : File{\rightarrow}read() \Leftarrow \%\grave{}, \theta)^{\infty}(\grave{}\%\grave{}, \theta)(\sim \grave{}\{\%\} : File{\rightarrow}read() \Leftarrow \%\grave{}, \theta)^{\infty}$$
$$(\sim \grave{}\{\%\} : File{\rightarrow}read() \Leftarrow \%\grave{}, \theta)^{\infty}(\grave{}\%\grave{}, \theta)(\sim \grave{}\{\%\} : File{\rightarrow}read() \Leftarrow \%\grave{}, \theta)^{\infty}$$
$$(\sim \grave{}\{\%\} : File{\rightarrow}read() \Leftarrow \%\grave{}, \theta)^{\infty}(\grave{}\%\grave{}, \theta)(\sim \grave{}\{\%\} : File{\rightarrow}read() \Leftarrow \%\grave{}, \theta)^{\infty}$$

Finally, we can use an exponent to indicate several instances of a policy concatenated together:

$$LimitFileReads(Integer\ max) : OneFileRead()^{max}$$

Now that some PoCo policies and combinators have been demonstrated, we proceed to detail the entirety of the language's features in the next section. Although most of the important features of the language have been appeared in the previous examples, a more general treatment than some instances of use is necessary.

5.2 Syntax Definition

The syntax of PoCo is now presented as a context-free grammar with commentary beside or below the productions. These comments indicate the purpose or meaning of the nonterminals and their rules.

ESC ::= `[]?@_%$

Escape characters have special meaning

SYM ::= $\overline{ESC}$

Non-escape characters are symbols

| $\backslash ESC$

Escape characters are symbols when preceded by a backslash

ID ::= $[a-zA-Z][a-zA-Z0-9]^*$

Identifiers

QID ::= $ID \mid QID.ID$

Qualified identifiers

$FIELDLIST$ ::= $ID : R$

Used to express object fields as comma separated list

$|FIELDLIST, ID : R$

$OBJECT$ ::= $\{QID\} : QID$

Objects can refer to a static field by its qualified identifier

$| \{R\} : TYP$

Objects that are wrappers for simple data have no fields

$| \{FIELDLIST\} : QID$

General objects are specified by their field values and a class name

$| \mathit{null}$

The null object is specified plainly

$| ID$

An object variable

$| \{0 : OBJECT, 1 : OBJECT, \ldots, n : OBJECT\} : \mathit{Array}$

Arrays are objects whose fields are sequential natural numbers

TYP ::= $\mathit{Byte} \mid \mathit{Short} \mid \mathit{Integer} \mid \mathit{Long} \mid \mathit{Float}$

$\mid \mathit{Double} \mid \mathit{Boolean} \mid \mathit{Char} \mid \mathit{String}$

The object types without fields

R REs that are used both in input REs and in signed REs

::= SYM a single literal or escaped symbol

$| RR$ Concatenation

$| [R]$ Grouping

$| R|R$ Switching

$| R^*$ Repetition 0 or more times

$| R^+$ Repetition 1 or more times

$| \sim R$ Complementation

$| R?$ Optional (i.e., 0 or 1 occurrences of R)

| % Multi-character wildcard, equivalent to SYM^*

| $\$ID$ Gets replaced with bound variable

| $@ID[R]$ Binds variable

IRE REs used to match against input events

$::= \grave{}R \Leftarrow R\grave{}$

Matches an action caused by a result

$| \grave{}R \Rightarrow R\grave{}$

Matches a result caused by an action

$| \grave{}R \Leftrightarrow R$

Matches an input action or result

$| \sim \grave{}R \Leftarrow R\grave{}$

Matches everything except an action caused by result

$| \sim \grave{}R \Leftrightarrow R\grave{}$

Matches everything except an input action or result

$SREBOP$ Binary operations for SREs

$::= \pm$ Optimistic Union

$| \mp$ Pessimistic Union

$| \top$ Disjunction

$| \bot$ Conjunction

$SREUOP ::= \sim$Unary negation operation for SREs

SRE Signed REs

$::= \theta$

The SRE where everything is in the neutral set

$| + \grave{}R\grave{}$

Positive RE

$| - \grave{}R\grave{}$

Negative RE

$| [SRE]$

Grouping of SREs

$| ID$

An identifier bound to an SRE

$| SRE\ SREBOP\ SRE$

Apply binary operation on SREs

$| SREUOP\ SRE$

Apply unary operation on SREs

$| +\!+\!`R`$

Positively signs R in output and follows the abstract exchange in which it occurs with $(\sim `R \Rightarrow \%`, +`R`)^{\infty}$.

$STATEBINDING$ ::= $@ID[R]$Used for binding variables outside of REs

$BEXCH$ Exchanges for base policies

$::= (IRE, SRE)$

Standard exchange

$| WaitForAction\ R$

Equivalent to $(\sim `R \Leftarrow \%`, -`R`)^{\infty}$.

$| WaitForResult\ R$

Equivalent to $(\sim `R \Rightarrow \%`, \theta)^{\infty}$.

$| STATEBINDING$

Statebinding at any point, but not counted as an exchange

$| PINST$

A policy applied to arguments is an execution e.g., $OneFileRead()$

$PINST$ The instantiation of a policy

::= $ID()$

A policy defined without parameters

| $ID(ARGS)$

A policy defined with some parameters

$ARGS$ An argument list

::= $OBJECT$

A single argument

| $ARGS, OBJECT$

A list of arguments, plus one more

$BEXECUTION$ Base policy execution

::= $BEXCH$

Base list of exchanges

| $BEXECUTION\ BEXECUTION$

Concatenation

| $[BEXECUTION]$

Grouping

| $BEXECUTION \mid BEXECUTION$

Switching

| $BEXECUTION^{\infty}$

Repeat 0 or more times

| $BEXECUTION^{OBJECT}$

An exact number of repetitions

$$| \bigcup_{SRE}^{+} [BEXECUTION]$$

Adds $\pm$ SRE to every result in BEXECUTION

$$| \bigcup_{SRE}^{-} [BEXECUTION]$$

Adds $\mp$ SRE to every result in BEXECUTION

$SPCOND$ A condition on subpolicy outputs

$::= ID \rightarrow SRE$

Bind ID to a signed regular language (always succeeds)

$| R\ ELEM\ SRE$

Test whether R is a subset of a component of ID

$| SRE = SRE$

$| SRE \neq SRE$

Signed regular equality tests for subpolicy output

$| |SRE| = \infty$

$| |SRE| \neq \infty$

Test whether a signed regular language has an infinite component

$SPCONDS$ List of conditions on subpolicy outputs

$::= SPCOND$

A single condition

$| SPCONDS,\ SPCOND$

A list of conditions, plus an additional one

$ELEM$ Binary signed membership ($\subseteq$) test

$::= R \in^{+} ID \mid R \notin^{+} ID \mid R \in^{-} ID \mid R \notin^{-} ID \mid R \in^{\pm} ID \mid R \notin^{\pm} ID$

$CXCH$ The contents of a combinator definition

$::= (IRE, \_, SRE)$

Combinator exchanges without subpolicy conditions

$| (IRE, (SPCONDS), SRE)$

Combinator exchanges with subpolicy conditions

$| STATEBINDING$

Statebinding at any point, but not counted as an exchange

$| ID$

A policy variable (e.g., in the `Cat` combinator)

$CEXECUTION$ Combinator Execution

$::= CEXCH$

Exchange/statebinding

$| CEXECUTION\ CEXECUTION$

Concatenation

$| [CEXECUTION]$

grouping

$| CEXECUTION \mid CEXECUTION$

Switching

$| CEXECUTION^{\infty}$

0 or more repetitions

$| CEXECUTION^{OBJECT}$

An exact number of repetitions

$| \overset{+}{\underset{SRE}{\bigcup}} [CEXECUTION]$

Adds ± SRE to every result in CEXECUTION

$$| \bar{\bigcup_{SRE}} [CEXECUTION]$$

Adds $\mp$ SRE to every result in CEXECUTION

$PARAMLIST$ List of parameters for parameterized policies

$::= QID\ ID$

$| PARAMLIST, QID\ ID$

$PPOL$ Complete PoCo policy without auxiliary methods

$::= ID(PARAMLIST)^{\infty} : CEXECUTION$

Repeating combinator policy with parameters

$| ID(PARAMLIST)^{\infty} : BEXECUTION$

Repeating base policy with parameters

$| ID(PARAMLIST) : CEXECUTION$

Non-repeating combinator policy with parameters

$| ID(PARAMLIST) : BEXECUTION$

Non-repeating base policy with parameters

$| ID()^{\infty} : CEXECUTION$

Repeating combinator policy with no parameters

$| ID()^{\infty} : BEXECUTION$

Repeating base policy with no parameters

$| ID() : CEXECUTION$

Non-repeating combinator policy with no parameters

$| ID() : BEXECUTION$

Non-repeating base policy with no parameters

$POLICY$ A complete policy declaration

$::= PPOL$

A policy with no auxiliary methods

$\mid PPOL\ METHODS$

A policy with auxiliary methods

$METHOD$ An auxiliary method declaration

$::= \ddots\ METHOD\_DECL$

A Java method declaration (Java's grammar not reproduce here)

$METHODS$ A sequence of auxiliary method declarations

$::= METHOD$

$\mid METHODS\ METHOD$

Now that PoCo's syntax has been fully introduced, we proceed to apply its constructs in the next section to implement a complex policy. The ability to define auxiliary methods along with PoCo policies allows tedious computations to be consolidated into a single action. This is used in the case study to emulate Polymer's `handleResult` method, in which policies can specify an arbitrary algorithm to execute whenever their suggestion is followed.

5.3 Case Study: Polymer's Email Policy

This section establishes that PoCo is an expressive language by implementing the sophisticated security policy for an email client that was originally written using Polymer [3, 4]. We first implement the base policies, which are self-contained modules that address particular security concerns. We then implement each of the super policies, which combine or augment the behavior of one or more policies. Some of the policy figures require line wrapping to overcome the limited width of this dissertation's format. Every line that is a

continuation of the previous one begins with a ↪. Wrapped lines are also indented at least two levels deeper than the original line to further distinguish wrapping from nesting.

5.3.1 Base Policies

We start this case study by implementing each of the base policies, which detail how the monitor should respond to specific security-relevant events. Each of these policies highlights a certain facet of email-client behavior that is a potential source of danger. Many of these potentially dangerous behaviors are not particular to email clients, so several of these policies are relevant to a variety of applications.

We begin with the simplest policies, which only involve a single abstract exchange. Figure 5.2 shows the PoCo versions of the `Trivial`, `DisSysCalls`, `Reflection`, and `NoOpenClassFiles` policies, collectively. The `Trivial` policy outputs θ for any input, so it has no effect on executions. Even so, it is useful as a subpolicy for combinators that select which of its subpolicies' responses to output, effectively toggling the enforcement of a subpolicy on and off.

$$Trivial()^{\infty} : (`\%`, \theta)$$
$$DisSysCalls()^{\infty} : (`\%`, -`Runtime.exec(\%)`)$$
$$Reflection()^{\infty} : (`\%`, -`\{Name : \%PoCo\%\} : Method{\rightarrow}invoke(\%)`)$$
$$NoOpenClassFiles()^{\infty} : (`\%`, -`File.init(\{\%.class\} : String)`)$$

Figure 5.2. Implementation of some simple Polymer base policies in PoCo.

Each of the remaining policies mentioned above is part of the policy branch that prevents the application from circumventing the security monitor, which should be a part of every policy tree. `DisSysCalls` prevents applications from using `Runtime.exec` to execute arbitrary strings, which could be used to wrap and disguise dangerous actions. The `Reflection` policy prevents the application from invoking `Method` objects (which are obtained using reflection) that belong to the policy-enforcement mechanism. This is done by prohibiting the invocation of `Method`s whose fully qualified name is prefixed by the package name belonging

to the mechanism code. Because our mechanism is not yet implemented, we will simply prohibit the invocations of `Method`s whose name contains "PoCo". Finally, `NoOpenClassFiles` prevents the application from accessing compiled code that could be executed by the Java runtime. This is done by prohibiting `File` objects from being instantiated (via the special `init` method) on paths ending in ".class".

The `AllowOnlyMIME` policy, shown in Figure 5.3, restricts the ports that a network socket can be opened on. Because the application is an email client, it should only communicate over the network on ports that are associated with email protocols. The IMAP, SMTP, and POP3 protocols use ports 143, 25, and 110, respectively. IMAP and POP3 transmissions can be secured using SSL, in which case IMAP uses port 993 and POP3 uses port 995. Because sockets can be opened in a multitude of ways, Polymer uses an *abstract action* called `NetworkOpens` that matches all such actions. PoCo, as it is currently defined, lacks an analogue of abstract actions, but is able to enumerate all relevant actions in a single regular expression.

Figure 5.4 contains the `Attachments` policy, which intercepts actions that would create a file having a forbidden extension. For the email client, this could happen if the user down-

$$
\begin{aligned}
&AllowOnlyMIME()^{\infty}: \\
&\quad @ports[`\{\sim [143|993|25|110|995]\} : Integer`] \\
&\quad (`\%`, -`[\{\%\} : mail.[imap.IMAPStore|pop3.POP3Store|smtp.SMTPTransport] \rightarrow \\
&\qquad\qquad \hookrightarrow \quad protocolConnect(\%, \$ports\%) \\
&\qquad \hookrightarrow \quad |java.net.Socket.init(\%, \$ports\%) \\
&\qquad \hookrightarrow \quad |\{\%\} : DatagramSocket \rightarrow send(\{port : \$ports\} : DatagramPacket) \\
&\qquad \hookrightarrow \quad |\{port : \$ports\} : java.net.MulticastSocket \rightarrow [leave|join]Group(\%) \\
&\qquad \hookrightarrow \quad |\{\%\} : java.net.MulticastSocket \rightarrow \\
&\qquad\qquad \hookrightarrow \quad send(\{port : \$ports\} : DatagramPacket, \%)]`)
\end{aligned}
$$

Figure 5.3. PoCo version of Polymer's `AllowOnlyMIME` policy.

loads an attachment. The $WaitForAction$ `$fileWrite` line outputs θ and repeats until the input is an action matching the $fileWrite$ pattern. The policy issues a warning via a popup window, where the user can choose to allow or cancel the download. This policy uses "statebinding" to bind ext to a regular expression matching all of the forbidden file extensions before the first abstract exchange. This allows us to conveniently maintain the forbidden extensions in one place for the whole policy. The $\bigcup^{+}_{-`\$fileWrite`}[\ldots]$ construct allows us to prohibit all file-writing actions in each abstract exchange within the brackets, except when we positively sign such an action (e.g., when $+$`$call` is output). This is because $\bigcup^{+}$ uses the $\pm$ operator to combine each output with the prohibited events. This also means that the θ output of the last abstract exchange in the policy is really $-$`$fileWrite`.

The `ConfirmAndAllowOnlyHTTP` policy, as displayed in Figure 5.5, alerts the user whenever the target attempts to establish an HTTP network connection, and prevents other types of network connections. This policy is essentially a cross between `AllowOnlyMIME` and `Attachments`.

The purpose of the `ClassLoaders` policy is to prevent the target from loading arbitrary classes at runtime. The policy intercepts instantiations of `ClassLoader` objects in order to inspect the call stack and determine whether a trusted component is loading the class or not. If so, then it is allowed; otherwise, it is prohibited. This policy makes use of arrays: the call stack obtained from an exception is one. Whereas Polymer iterates through this array, checking the prefix of each entry against each trusted package name, PoCo can express a pattern that matches when any array element has such a prefix. When the target attempts to load a class, we output $-$`%`, which prohibits all events, to emulate Polymer's `HaltSug` output. This policy's specification appears in Figure 5.6.

`InterruptToCheckMem` instantiates an interrupt-generating object, then matches against the interrupts it generates in order to regularly check what percentage of memory is being used. Because this policy wants to run the interrupt generator as soon as possible, it promotes doing so in lieu of the first input action. After the generator is running, the

policy then requires the execution of the initial action. After that, it outputs θ on all inputs until an interrupt is input. If memory usage is within tolerance, θ is repeatedly output. If memory usage exceeds the given limit, then a popup window displays a warning, and it continues outputting θ for all future inputs. That is, the popup window appears only once:

$Attachments()^{\infty}$:

$@ext[`.[exe|vbs|hta|mdb|bad]`]$

$@fileWrite[`\{name : \{\%\$ext\} : String\} : File \rightarrow createNewFile()$

$\hookrightarrow \quad |File.createTempFile(\{\%\} : String, \{\%\$ext\} : String, \%)$

$\hookrightarrow \quad |\{\%\} : File \rightarrow renameTo(\{name : \{\%\$ext\} : String\} : File \rightarrow)$

$\hookrightarrow \quad |FileOutputStream.init(\{\%\$ext\} : String, \%)$

$\hookrightarrow \quad |FileOutputStream.init(\{name : \{\%\$ext\} : String\} : File)$

$\hookrightarrow \quad |RandomAccessFile.init(\{\%\$ext\} : String, \%)$

$\hookrightarrow \quad |RandomAccessFile.init(\{name : \{\%\$ext\} : String\} : File, \%)`]$

$WaitForAction\ `\$fileWrite`$

$\bigcup^{+}_{-`\$fileWrite`}$

[

(`$@call[\$fileWrite] \Leftarrow \%`, +`$@confirm[showConfirmDialog(null,$

$\hookrightarrow \quad$ {The target is creating a file via:$\$call$. This is a

$\hookrightarrow \quad$ dangerous file type. Do you want to create this file?} : $String$,

$\hookrightarrow \quad$ {Security Question} : $String$,

$\hookrightarrow \quad \{JOptionPane.YES\_NO\_OPTION\} : Integer)]`)$

[(`$\$confirm \Rightarrow \{JOptionPane.OK\_OPTION\} : Integer`, +`$\$call`)

| (`%`, θ)]

]

Figure 5.4. PoCo version of Polymer's `Attachments` policy.

when memory usage is first observed to be over the limit. Observe that the target is not terminated for using excessive memory. This policy is specified in Figure 5.7.

The `IncomingMail` policy, shown in Figure 5.8, augments the email client's handling of received emails by doing each of the following:

$ConfirmAndAllowOnlyHTTP()^{\infty}$:

$@ports[`\{[80|443]\} : Integer`]$

$@netCon[`\{\%\} : mail.[imap.IMAPStore|pop3.POP3Store|smtp.SMTPTransport]\rightarrow$

$\hookrightarrow \quad protocolConnect(\%, \$ports\%)$

$\hookrightarrow \quad |java.net.Socket.init(\%, \$ports\%)$

$\hookrightarrow \quad |\{\%\} : DatagramSocket\rightarrow send(\{port : \$ports\} : DatagramPacket)$

$\hookrightarrow \quad |\{port : \$ports\} : java.net.MulticastSocket\rightarrow [leave|join]Group(\%)$

$\hookrightarrow \quad |\{\%\} : java.net.MulticastSocket\rightarrow$

$\hookrightarrow \quad send(\{port : \$ports\} : DatagramPacket, \%)`]$

$WaitForAction\ `\$netCon`$

$\bigcup^{+}_{-`\$netCon`}$

[

(`$@call[\$netCon]\Leftarrow\%`, +`$@confirm[showConfirmDialog(null,$

$\hookrightarrow \quad \{$The program is attempting to make an HTTP connection

$\hookrightarrow \quad$ via:$\$call$. Do you want to allow this connection?$\} : String,$

$\hookrightarrow \quad \{$Security Question$\} : String,$

$\hookrightarrow \quad \{JOptionPane.YES\_NO\_OPTION\} : Integer)]`)$

[(`$\$confirm\Rightarrow\{JOptionPane.OK\_OPTION\} : Integer`, +`$\$call`)

| (`%`, θ)]

]

Figure 5.5. PoCo version of Polymer's `ConfirmAndAllowOnlyHTTP` policy.

1. Checks each message sender's address against a file of trusted addresses, and marks messages from untrusted addresses by prepending "SPAM? - " to the subject line.
2. Imposes a limit on the length of the subject line.
3. Issues a warning when new emails from untrusted addresses contain attachments.
4. Logs all messages in a local file.

Items 1-3 above are handled by the `spamifySubject` action, and Item 4 is performed by the `log` action.

The `OutgoingMail` policy is specified in Figure 5.9. When an email is to be sent out, this policy does the following:

1. Logs the outgoing message.
2. Pops up a confirmation window listing the email's recipients.
3. Blind copies (BCCs) the mail to a given address for backup purposes.

$$
\begin{aligned}
&ClassLoaders()^{\infty}: \\
&\quad WaitForAction\ \grave{}ClassLoader.init(\%)\grave{} \\
&\quad \bigcup\nolimits^{+}_{-\grave{}ClassLoader.init(\%)\grave{}} \\
&\qquad [\\
&\qquad\quad (@call[\grave{}ClassLoader.init(\%)] \Leftarrow \%\grave{}, ++\ \grave{}Exception.init()\grave{}) \\
&\qquad\quad (\grave{}Exception.init() \Rightarrow @ex\{\%\}\grave{}, ++\ \grave{}\{\$ex\} : Exception \rightarrow getStackTrace()\grave{}) \\
&\qquad] \\
&\quad [\ (\grave{}\{\$ex\} : Exception \rightarrow getStackTrace() \Rightarrow \\
&\qquad\qquad \hookrightarrow\ \ \{\%[0-9]^{+} : \{[java.|javax.|org.apache.|com.sun.|sun.]\%\} : Array\grave{}, \\
&\qquad\qquad \hookrightarrow\ \ +\grave{}\$call\grave{}) \\
&\quad |\ (\grave{}\{\$ex\} : Exception \rightarrow getStackTrace() \Rightarrow \%\grave{}, -\grave{}\%\grave{})\ ]
\end{aligned}
$$

Figure 5.6. PoCo version of Polymer's `ClassLoaders` policy.

4. Concatenates contact information to the email text.

The message logging and confirmation popup is performed by the auxiliary `log` and `confirm` actions, respectively.

5.3.2 Super Policies

Now we need to be able to appropriately combine the base policies such that our final composition behaves like the original Email policy in Polymer. The `IsClientSigned`

$$
\begin{aligned}
&InterruptToCheckMem(Double\ maxPercent, Long\ interval)^{\infty}: \\
&\quad (\texttt{`}@first[\%] \Leftarrow \%\texttt{`}, ++\ \texttt{`}mail.interrupts.InterruptsGen.init(\$interval)\texttt{`}) \\
&\quad (\texttt{`}\% \Rightarrow @ig[\%]\texttt{`}, ++\ \texttt{`}\$ig{\rightarrow}start()\texttt{`}) \\
&\quad (\texttt{`}\%\texttt{`}, ++\ \texttt{`}\$first\texttt{`}) \\
&\quad [\ (\texttt{`}mail.interrupts.InterruptGen.interrupt() \Rightarrow \%\texttt{`}, ++\ \texttt{`}Runtime.getRuntime()\texttt{`}) \\
&\qquad (\texttt{`}\% \Rightarrow @run[\%]\texttt{`}, ++\ \texttt{`}run.totalMemory()\texttt{`}) \\
&\qquad (\texttt{`}\% \Rightarrow @totalM[\%]\texttt{`}, ++\ \texttt{`}run.maxMemory()\texttt{`}) \\
&\qquad (\texttt{`}\% \Rightarrow @maxM[\%]\texttt{`}, ++\ \texttt{`}div(\$totalM, \$maxM)\texttt{`}) \\
&\qquad (\texttt{`}\% \Rightarrow @decPerc[\%]\texttt{`}, ++\ \texttt{`}mult(\$decPerc, \{100\} : Double)\texttt{`}) \\
&\qquad (\texttt{`}\% \Rightarrow @perc[\%]\texttt{`}, ++\ \texttt{`}gt(\$perc, \$maxPercent)\texttt{`}) \\
&\qquad [\ (\texttt{`}\% \Rightarrow \{true\} : Boolean\texttt{`}, ++\ \texttt{`}showConfirmDialog(null, \\
&\qquad\qquad \hookrightarrow\ \{\text{More than \$maxPercent\% of the memory} \\
&\qquad\qquad \hookrightarrow\ \text{available to the VM has been consumed}\} : String, \\
&\qquad\qquad \hookrightarrow\ \{\text{Warning}\} : String, \\
&\qquad\qquad \hookrightarrow\ \{JOptionPane.WARNING\_MESSAGE\} : Integer)\texttt{`}) \\
&\qquad\quad (\texttt{`}\%\texttt{`}, \theta)^{\infty} \\
&\qquad |\ (\texttt{`}\%\texttt{`}, \theta)\ ] \\
&\quad |\ (\texttt{`}\%\texttt{`}, \theta)\ ]^{\infty}
\end{aligned}
$$

Figure 5.7. PoCo version of Polymer's `InterruptToCheckMem` policy.

superpolicy, seen in Figure 5.10, selects which subpolicy to enforce depending on whether the target is signed with a certificate. The left-hand subpolicy is expected to be less restrictive than that on the right. The idea is that a cryptographic signature establishes a level of trust, so we can switch to enforcing a more lax policy on such a target. While it is unknown whether the target is signed, we presume that it is not and enforce $p2$, the more stringent policy. Once it is determined whether the target is signed (i.e., when $isSigned()$ is executed), we enforce the appropriate subpolicy for the remainder of the execution.

$$
\begin{aligned}
&IncomingMail()^{\infty}: \\
&\quad @getMail[\texttt{`}com.sun.mail.imap.IMAPFolder.expunge() \\
&\qquad \hookrightarrow |com.sun.mail.imap.IMAPFolder.fetch(\%) \\
&\qquad \hookrightarrow |com.sun.mail.imap.IMAPFolder.getMessage(\%) \\
&\qquad \hookrightarrow |com.sun.mail.imap.IMAPFolder.getMessageByUID(\%) \\
&\qquad \hookrightarrow |com.sun.mail.imap.IMAPFolder.search(\%) \\
&\qquad \hookrightarrow |com.sun.mail.pop3.POP3Folder.expunge() \\
&\qquad \hookrightarrow |com.sun.mail.pop3.POP3Folder.fetch(\%) \\
&\qquad \hookrightarrow |com.sun.mail.pop3.POP3Folder.getMessage(\%)\texttt{`}] \\
&\quad WaitForAction\ \texttt{`}[\$getMail\,|\,\{\%\}:Message{\rightarrow}getSubject()]\texttt{`} \\
&\quad \bigcup^{+}_{-\texttt{`}\$getMail\texttt{`}\mp-\texttt{`}\{\%\}:Message\rightarrow getSubject()\texttt{`}}\ [\\
&\qquad\quad (\texttt{`}\$getMail \Rightarrow @result\{\%\}\texttt{`}, ++\texttt{`}log(\$result)\texttt{`}) \\
&\qquad\quad |\ (\texttt{`}\{@message[\%]\}:Message{\rightarrow}getSubject() \Rightarrow \%\texttt{`}, \\
&\qquad\qquad\quad \hookrightarrow\quad ++\texttt{`}spamifySubject(\$message)\texttt{`})\ ] \\
&\because \textbf{public static void}\ \log(Message\ m)\ \{\ldots\} \\
&\because \textbf{public static String}\ \text{spamifySubject}(Message\ m)\ \{\ldots\}
\end{aligned}
$$

Figure 5.8. PoCo version of Polymer's `IncomingMail` policy. See Appendix A for the auxiliary method definitions.

The `Audit` superpolicy, depicted in Figure 5.11, maintains a log of all input actions and its subpolicy's response to them. When the first action is input, it is bound to *act*, the subpolicy response is bound to *out* (via $out \leftarrow p$), and the log file is opened. Once the log file is opened, *act* and *out* are written to the log. After updating the log, `Audit` has

$$
\begin{array}{l}
OutgoingMail(String\ ContactInfo)^{\infty}: \\
\quad @sendMail[\grave{}\,javax.mail.Transport.send(@msg[\{\%\}:Message]) \\
\qquad \hookrightarrow \ \ |javax.mail.Transport.send(@msg[\{\%\}:Message],\%) \\
\qquad \hookrightarrow \ \ |com.sun.mail.smtp.SMTPTransport. \\
\qquad\quad \hookrightarrow \ \ sendMessage(@msg[\{\%\}:Message],\%)\grave{}\,] \\
\quad WaitForAction\ \grave{}\$sendMail\grave{} \\
\quad \bigcup^{+}_{-\grave{}\$sendMail\grave{}}\ [\\
\qquad (\grave{}\$sendMail \Rightarrow \%\grave{}, ++\grave{}\,log(\$msg)\grave{}) \\
\qquad (\grave{}\%\grave{}, ++\grave{}\,confirm(\$msg)\grave{}) \\
\qquad [\ (\sim \grave{}\%{\Rightarrow}\{JOptionPane.OK\_OPTION\}:Integer\grave{}, +\grave{}\,null\grave{}) \\
\qquad |\ (\grave{}\%{\Rightarrow}\{JOptionPane.OK\_OPTION\}:Integer\grave{}, \\
\qquad\quad \hookrightarrow \ \ ++\grave{}\,\$msg{\rightarrow}AddBCC(\{\text{“}user@domain''\}:String)\grave{}) \\
\qquad\ (\grave{}\%{\Rightarrow}@msg\{\%\}\grave{}, ++\grave{}\,\$msg{\rightarrow}getContent()\grave{}) \\
\qquad\ (\grave{}\%{\Rightarrow}@content\{\%\}\grave{}, ++\grave{}\,strCat(\$content,\$ContactInfo)\grave{}) \\
\qquad\ (\%{\Rightarrow}@content\{\%\}\grave{}, ++\grave{}\,\$msg{\rightarrow}setContent(\$content)\grave{}) \\
\qquad\ (\grave{}\%{\Rightarrow}@msg\{\%\}\grave{}, ++\grave{}\,mail.SendMail(\$msg)\grave{}) \\
\qquad\ (\grave{}\%{\Rightarrow}@result\{\%\}\grave{}, +\grave{}\,\$result\grave{}) \\
\qquad\ ]\ ] \\
\because \textbf{public static void}\ \log(Message\ m)\ \{\ldots\} \\
\because \textbf{public static Integer}\ \mathrm{confirm}(Message\ m)\ \{\ldots\}
\end{array}
$$

Figure 5.9. PoCo version of Polymer's `OutgoingMail` policy. See Appendix A for the auxiliary method definitions.

to determine how the monitor would have proceeded if the subpolicy was being enforced on its own. This is achieved by matching against the input and signed regular output in the same manner as the MRA algorithm in Section 5.1. This algorithm entails determining whether the subpolicy output positively signs infinitely-many events, which is performed by the cardinality-testing expression $|\ldots| \neq \infty$. After emitting the appropriate output, the subpolicy's response is output while results are input. When the next action is input, the bindings are updated and we loop back to the logging step.

Polymer's `Conjunction` and `Disjunction` combinators generally output the "most restrictive" and "least restrictive" subpolicy output, respectively. However, both of them give

$$
\begin{array}{l}
IsClientSigned(Policy\ p1, Policy\ p2): \\
\quad (\sim \text{`}isSigned() \Rightarrow \%\text{`}, \_, p2 \mp +\text{`}isSigned()\text{`})^{\infty} \\
\quad [\ (\text{`}isSigned() \Rightarrow \{true\} : Boolean\text{`}, \_, p1) \\
\qquad (\text{`}\%\text{`}, \_, p1)^{\infty} \\
\quad |\ (\text{`}isSigned() \Rightarrow \{false\} : Boolean\text{`}, \_, p2) \\
\qquad (\text{`}\%\text{`}, \_, p2)^{\infty} \\
\quad] \\
\because \textbf{private static Boolean}\ isSigned()\{ \\
\qquad Enumeration\ e = PoCo.getJarFile().entries(); \\
\qquad \textbf{while}(e.hasMoreElements())\{ \\
\qquad\quad Certificate[]\ ca = ((JarEntry)e.nextElement()).getCertificates(); \\
\qquad\quad \textbf{if}\ (ca! = null\ \&\&\ ca.length > 0\ \&\&\ ca[0]! = null) \\
\qquad\qquad \textbf{return}\ true; \\
\qquad \} \\
\qquad \textbf{return} false; \\
\quad \}
\end{array}
$$

Figure 5.10. PoCo version of Polymer's `IsClientSigned` combinator.

highest priority to `InsSug`s the lowest priority to `IrrSug`s. In order to specify these policies in a form that is truest to the Polymer versions, we establish an interpretation of Polymer's suggestions with respect to signed regular languages. For input event e_{in}, each suggestion has the following signed regular interpretation:

$$
\begin{array}{l}
Audit(Policy\ p, String\ f):\\
\quad (\texttt{`}@act[\%] \Leftarrow \%\texttt{`}, (out \leftarrow p), ++\texttt{`}fopen(\$f)\texttt{`})\\
\quad (\texttt{`}\% \rightarrow @ps[\%], \_, ++\texttt{`}log(\$ps, \$out, \$act)\texttt{`})\\
\quad [\ [\ (\texttt{`}\%\texttt{`}, (|out\bot + \texttt{`}\%(\%)\texttt{`}\bot + \texttt{`} \sim \$act\texttt{`}| \neq \infty), out)\\
\quad\ \ |\ (\texttt{`}\%\texttt{`}, (\texttt{`}\$act\texttt{`} \in^{+} out), +\texttt{`}\$act\texttt{`})\\
\quad\ \ |\ (\texttt{`}\%\texttt{`}, (|out\bot + \texttt{`}\{\%\} : \%\texttt{`}| \neq \infty), out)\\
\quad\ \ |\ (\texttt{`}\%\texttt{`}, (\texttt{`}\$act\texttt{`} \notin^{\pm} out), +\texttt{`}\$act\texttt{`})\\
\quad\ \ |\ (\texttt{`}\%\texttt{`}, \_, out)\ ]\\
\quad (\texttt{`}\% \Rightarrow \%\texttt{`}, \_, p)^{\infty}\\
\quad (\texttt{`}@act[\%] \Leftarrow \%\texttt{`}, (out \leftarrow p), ++\texttt{`}log(\$ps, \$out, \$act)\texttt{`})\ ]^{\infty}\\
\because \textbf{private static}\ \text{PrintStream}\ fopen(String\ fn)\{\\
\quad\ \textbf{return new}\ PrintStream(\\
\quad\quad \textbf{new}\ BufferedOutputStream(\\
\quad\quad\quad \textbf{new}\ FileOutputStream(fn)),\\
\quad\quad true);\\
\quad \}\\
\because \textbf{private static}\ \text{void}\ log(PrintStream\ ps, SRE\ s, Action\ a)\{\\
\quad\ ps.println(\text{“On trigger action ”} + a.toString());\\
\quad\ ps.println(\text{“Subpolicy output: ”} + s.toString());\\
\quad\ ps.println(\text{“}- - - - - - - - - - - - - - - - - - - -\backslash\text{n”});\\
\quad \}
\end{array}
$$

Figure 5.11. PoCo version of Polymer's `Audit` policy.

1. `Irrelevant` $= \theta$
2. `OK` $= +`e_{in}`$
3. `Insert(a)` $(a \neq e_{in}) = +`a`$
4. `Suppress` $= -`e_{in}`$
5. `Replace(r)` $(r \neq e_{in}) = +`r`$
6. `Halt` $= -`\%`$

Although insertion and replacement correspond to very similar signed regular languages, we can use the regular expression `` `%(%)` `` to match actions and `` `{%}:%` `` to match results. Therefore, sifting out a language's positive or negative actions or results is simple, and insertion can be distinguished from replacement.

The `Conjunction` combinator, whose PoCo listing appears in Figure 5.12, prioritizes the various suggestion types, and outputs the subpolicy output with the highest priority. This policy makes use of the `` `a ⇔ r` `` pattern, which matches the input, whether it is an action or result, against the appropriate regular expression in the pattern. Because only one of them will match, it makes sense to bind the same variable on both sides of the ⇔ symbol. The $\perp$ operation is used when subpolicies both output insertions or both output replacements so only their common positive events are output. This is similar to how the Polymer version only considers two insertions and two replacements equal when their event parameters are, but preserves subpolicy prohibitions. Finally, there is a catch-call case at the end for outputs that do not fit squarely into one of the suggestion categories.

The `Disjunction` combinator, presented in Figure 5.15, is defined very similarly to `Conjunction`, except the interior cases of `Conjunction` (i.e., Halt down to OK) are considered in the opposite order, and the last case combines the subpolicy outputs with $\pm$ rather than $\mp$. The case where both subpolicies perform insertions still uses $\perp$, rather than $\top$, to combine the outputs, because the Polymer version only combines `InsSug`s when

$Conjunction(Policy\ p1, Policy\ p2)^{\infty}$:

\\Insertions

(`$@in[\%] \Leftrightarrow @in[\%]$`, ($p1\bot$ + `$\%(\%)$`$\bot$ + ` $\sim$ `$\$in$` $\neq \theta$,

$\hookrightarrow$ $p1\bot p2\bot$ + `$\%(\%)$`$\bot$ + ` $\sim$ `$\$in$` $\neq \theta$),

$\hookrightarrow$ $p1\bot p2$)

| (`$@in[\%] \Leftrightarrow @in[\%]$`, ($p1\bot$ + `$\%(\%)$`$\bot$ + ` $\sim$ `$\$in$` $\neq \theta$), $p1$)

| (`$@in[\%] \Leftrightarrow @in[\%]$`, ($p2\bot$ + `$\%(\%)$`$\bot$ + ` $\sim$ `$\$in$` $\neq \theta$), $p2$)

\\Halt

| (`$\%$`, ($p1 = -$`$\%$`), $p1$)

| (`$\%$`, ($p2 = -$`$\%$`), $p2$)

\\Suppression

| (`$\%$`, ($p1 = -$`$\$in$`), $p1$)

| (`$\%$`, ($p2 = -$`$\$in$`), $p2$)

\\Replacement

| (`$@in[\%] \Leftrightarrow @in[\%]$`, ($p1\bot$ + `$\{\%\} : \%$`$\bot$ + ` $\sim$ `$\$in$` $\neq \theta$,

$\hookrightarrow$ $p1\bot p2\bot$ + `$\{\%\} : \%$`$\bot$ + ` $\sim$ `$\$in$` $\neq \theta$),

$\hookrightarrow$ $p1\bot p2$)

| (`$@in[\%] \Leftrightarrow @in[\%]$`, ($p1\bot$ + `$\{\%\} : \%$`$\bot$ + ` $\sim$ `$\$in$` $\neq \theta$), $p1$)

| (`$@in[\%] \Leftrightarrow @in[\%]$`, ($p2\bot$ + `$\{\%\} : \%$`$\bot$ + ` $\sim$ `$\$in$` $\neq \theta$), $p2$)

\\OK

| (`$\%$`, ($p1 = +$`$\$in$`), $p1$)

| (`$\%$`, ($p2 = +$`$\$in$`), $p2$)

\\Can't reach this point with Polymer-like outputs

| (`$\%$`, $\_$, $p1 \mp p2$)

Figure 5.12. PoCo version of Polymer's `Conjunction` policy.

the inserted actions are equal. The case where both subpolicy outputs are replacements is handled similarly.

The `Dominates` superpolicy has two subpolicies, $p1$ and $p2$. `Dominates` simply outputs whatever $p2$ does if the input is irrelevant to $p1$; otherwise, it outputs whatever $p1$ does. That is, $p1$ "dominates" $p2$, where $p2$ only gets a say if $p1$ has no interesting response. This policy is shown in Figure 5.13.

$$
\begin{aligned}
&\mathit{Dominates}(\mathit{Policy}\ p1, \mathit{Policy}\ p2)^{\infty}: \\
&\quad (\grave{}\%\grave{}, (p1 = \theta), p2) \\
&\quad \mid (\grave{}\%\grave{}, \_, p1)
\end{aligned}
$$

Figure 5.13. PoCo version of Polymer's `Dominates` policy.

The `TryWith` superpolicy also has two subpolicies, $p1$ and $p2$. This combinator is presented in Figure 5.14. It outputs whatever $p1$ outputs, unless $p1$ suggests halting, suppression, or replacement. When it does, then the combinator outputs whatever $p2$ outputs. In other words, this superpolicy enforces $p1$ foremost, but falls back to $p2$ for inputs that $p1$ is opposed to. This superpolicy's behavior is similar to `Try{...}With{...}` constructs in general-purpose programming languages, in which execution proceeds to a secondary block of code when an error arises in the primary block.

$$
\begin{aligned}
&\mathit{TryWith}(\mathit{Policy}\ p1, \mathit{Policy}\ p2)^{\infty}: \\
&\quad (\grave{}\%\grave{}, (p1 = \theta), p1) \\
&\quad \mid (\grave{}\ @in[\%] \Leftrightarrow @in[\%]\grave{}, (+\grave{}\ \$in\grave{} = p1), p1) \\
&\quad \mid (\grave{}\ @in[\%] \Leftrightarrow @in[\%]\grave{}, (p1\bot + \grave{}\ \%(\%)\grave{}\ \bot + \grave{}\ \sim \$in\grave{} \neq \theta), p1) \\
&\quad \mid (\grave{}\%\grave{}, \_, p2)
\end{aligned}
$$

Figure 5.14. PoCo version of Polymer's `TryWith` policy.

$$
\begin{aligned}
&Disjunction(Policy\ p1, Policy\ p2)^{\infty} : \\
&\quad \backslash\backslash \text{Insertions} \\
&\quad (\grave{}\,@in[\%] \Leftrightarrow @in[\%]\grave{}, (p1 \perp + \grave{}\,\%(\%)\grave{}\perp + \grave{} \sim \$in\grave{} \neq \theta, \\
&\qquad\qquad \hookrightarrow \quad p1 \perp p2 \perp + \grave{}\,\%(\%)\grave{}\perp + \grave{} \sim \$in\grave{} \neq \theta), \\
&\qquad\qquad \hookrightarrow \quad p1 \perp p2) \\
&\quad | \ (\grave{}\,@in[\%] \Leftrightarrow @in[\%]\grave{}, (p1 \perp + \grave{}\,\%(\%)\grave{}\perp + \grave{} \sim \$in\grave{} \neq \theta), p1) \\
&\quad | \ (\grave{}\,@in[\%] \Leftrightarrow @in[\%]\grave{}, (p2 \perp + \grave{}\,\%(\%)\grave{}\perp + \grave{} \sim \$in\grave{} \neq \theta), p2) \\
&\quad \backslash\backslash \text{OK} \\
&\quad | \ (\grave{}\,\%\grave{}, (p1 = +\grave{}\,\$in\grave{}), p1) \\
&\quad | \ (\grave{}\,\%\grave{}, (p2 = +\grave{}\,\$in\grave{}), p2) \\
&\quad \backslash\backslash \text{Replacement} \\
&\quad | \ (\grave{}\,@in[\%] \Leftrightarrow @in[\%]\grave{}, (p1 \perp + \grave{}\,\{\%\} : \%\grave{}\perp + \grave{} \sim \$in\grave{} \neq \theta, \\
&\qquad\qquad \hookrightarrow \quad p1 \perp p2 \perp + \grave{}\,\{\%\} : \%\grave{}\perp + \grave{} \sim \$in\grave{} \neq \theta), \\
&\qquad\qquad \hookrightarrow \quad p1 \perp p2) \\
&\quad | \ (\grave{}\,@in[\%] \Leftrightarrow @in[\%]\grave{}, (p1 \perp + \grave{}\,\{\%\} : \%\grave{}\perp + \grave{} \sim \$in\grave{} \neq \theta), p1) \\
&\quad | \ (\grave{}\,@in[\%] \Leftrightarrow @in[\%]\grave{}, (p2 \perp + \grave{}\,\{\%\} : \%\grave{}\perp + \grave{} \sim \$in\grave{} \neq \theta), p2) \\
&\quad \backslash\backslash \text{Suppression} \\
&\quad | \ (\grave{}\,\%\grave{}, (p1 = -\grave{}\,\$in\grave{}), p1) \\
&\quad | \ (\grave{}\,\%\grave{}, (p2 = -\grave{}\,\$in\grave{}), p2) \\
&\quad \backslash\backslash \text{Halt} \\
&\quad | \ (\grave{}\,\%\grave{}, (p1 = -\grave{}\,\%\grave{}), p1) \\
&\quad | \ (\grave{}\,\%\grave{}, (p2 = -\grave{}\,\%\grave{}), p2) \\
&\quad \backslash\backslash \text{Can't reach this point with Polymer-like outputs} \\
&\quad | \ (\grave{}\,\%\grave{}, \_, p1 \pm p2)
\end{aligned}
$$

Figure 5.15. PoCo version of Polymer's `Disjunction` policy.

CHAPTER 6

SUMMARY AND CONCLUSION

It is possible to create tools that aid the principled specification of expressive, composable software-security policies. Existing graphical tools for security-policy specification are limited to particular security domains and cannot be used for general software-security policies. Existing policy-specification languages that are simple and compose well have limited expressiveness (e.g., access control), and more expressive languages are complicated and suffer in terms of composability. This dissertation provides two aids to combat these shortcomings:

1. PoliSeer—A graphical tool for specifying, visualizing, and modifying composable policies.

2. PoCo—A policy-specification language whose policies are specified simply, compose well, and are expressive.

PoliSeer users rely on policy-composition experts to distribute libraries of universally composable policies (written in a language like Polymer). PoliSeer users can then build complex policies by composing those expert-written policies in meaningful ways. For example, we have constructed complex email-client and PoliSeer policies (Chapter 3; Figures 2.6 and 3.4). In our experience, PoliSeer has been a great aid for quickly specifying and generating code to enforce complex policies built as compositions of simpler subpolicies.

We believe PoliSeer is useful, even for expert policy engineers, for clearly and conveniently visualizing complex policy trees. Moreover, PoliSeer's interface contains several considerations for conveniently modifying policies, such as policy replacement, branch-insertion points (BIPs), and branch deletions (Chapter 2; Figure 2.8). Thus, we view PoliSeer as an

integrated development environment (IDE) for security policies, providing policy engineers the same sorts of benefits that traditional IDEs provide software engineers (convenience of creating high-level specifications and visualizations to minimize errors in, or totally avoid, low-level programming tasks).

PoCo users specify software security policies in terms of input-output (to/from the monitor) event sequences. These sequences may be defined generally, using regular expressions to describe the matching inputs and combinations of signed regular expressions to express the desired, irrelevant, and prohibited outputs at once. These descriptive outputs compose well: operations for combining them satisfy a large number of algebraic properties (Chapter 4.1). These properties ease the policy-specification effort because policy hierarchies are less sensitive to organizational details (e.g., we can ignore the order of a commutative combinator's subpolicies).

Poco is also expressive, which was demonstrated by implementing a sophisticated policy on an email client (Chapter 5). However, implementing the policies in a way that emulates their behavior in Polymer is cumbersome because policy outputs are much more expressive in PoCo. That is, we have to go against our principles of directly combining the subpolicy outputs and try to determine if they correspond to certain types of Polymer suggestions. The most striking instances of this difference are the `Conjunction` and `Disjunction` combinators, each of which takes thirteen abstract exchanges to express. In contrast, rich conjunctive and disjunctive combinators can be specified in PoCo using a single abstract exchange by simply applying the $\bot$ and $\top$ operations on subpolicy outputs, respectively.

Because PoCo policies have a form akin to executions, policy authors are confronted explicitly with the sequences of behaviors that policies can exhibit. The explicitness of the policy definitions promotes thorough consideration of all possibilities at each stage, which should minimize programmer oversight. PoCo's declarative style makes literal combination of policies as data quite natural. Literal combination can be helpful for specifying more sophisitcated policies as sequences of small policies that contain only a few abstract exchanges.

The PoliSeer and PoCo efforts, though discussed separately, are also helpful together. Consider that literal combination could be used with a tool like PoliSeer visually specify some base policies. For example, a policy hierarchy using the concatenative combinator and the `OneFileRead` policy would constitute a visual representation as well as an implementation of the `LimitFileReads` policy (for a specific limit).

In the future, we plan to continue to study techniques and tools for managing policy composition. Our highest-priority future work involves implementing the PoCo language. From this implementation, we plan to experiment with static policy analyses, which may be helpful for detecting policy errors and for policy optimization.

LIST OF REFERENCES

[1] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[2] Lujo Bauer, Jarred Ligatti, and David Walker. A language and system for composing security policies. Technical Report TR-699-04, Princeton University, January 2004.

[3] Lujo Bauer, Jay Ligatti, and David Walker. Composing expressive run-time security policies. *ACM Transactions on Software Engineering and Methodology*. To appear.

[4] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

[5] Lujo Bauer, Jay Ligatti, and David Walker. Polymer: A language for composing run-time security policies, 2008. `http://www.cs.princeton.edu/sip/projects/polymer/`.

[6] R. Bhatti, M.L. Damiani, D.W. Bettis, and E. Bertino. Policy mapper: Administering location-based access-control policies. *Internet Computing, IEEE*, 12(2):38–45, March-April 2008.

[7] Carolyn A. Brodie, Clare-Marie Karat, and John Karat. An empirical study of natural language parsing of privacy policy rules using the SPARCLE policy workbench. In *Proceedings of the second symposium on Usable privacy and security*, pages 8–19, 2006.

[8] Glenn Bruns and Michael Huth. Access control via belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Trans. Inf. Syst. Secur.*, 14(1):9:1–9:27, June 2011.

[9] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–39, 2001.

[10] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag, Berlin, 2007.

[11] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-based access control for mobile code. In *ACM Conference on Computer and Communications Security*, 1998.

[12] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[13] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, 1999.

[14] Robert Gilmer. Commutative semigroup rings. chicago lectures in mathematics, 1984.

[15] K Glazek. A guide to the literature on semirings and their applications in mathematics and information sciences: with complete bibliography, kluwer acad. publ., dordrecht e.a, 2002.

[16] George A. Gratzer. *General lattice theory / by George Gratzer.* Academic Press, New York :, 1978.

[17] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):158–173, 2004.

[18] Philip Inglesant, M. Angela Sasse, David Chadwick, and Lei Lei Shi. Expressions of expertness: the virtuous circle of natural language for access control policy specification. In *Proceedings of the 4th symposium on Usable privacy and security*, pages 77–88, 2008.

[19] JavaCC, 2008. `https://javacc.dev.java.net/`.

[20] Clinton Jeffery, Wenyi Zhou, Kevin Templer, and Michael Brazell. A lightweight architecture for program execution monitoring. In *Program Analysis for Software Tools and Engineering (PASTE)*, 1998.

[21] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*, 1999.

[22] Yingsha Liao and Donald Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Trans. Softw. Eng.*, 18(11):969–978, 1992.

[23] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, February 2005.

[24] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, 2009.

[25] Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, September 2010.

[26] Daniel Lomsak and Jay Ligatti. Poliseer: A tool for managing complex security policies. In *International Conference on Trust Management (IFIP-TM)*, June 2010.

[27] Daniel Lomsak and Jay Ligatti. Poliseer: A tool for managing complex security policies. *JIP*, 19:292–306, 2011.

[28] A. Mayer, A. Wool, and E. Ziskind. Fang: a firewall analysis engine. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 177–187, 2000.

[29] Patrick McDaniel and Atul Prakash. Methods and limitations of security policy reconciliation. *ACM Trans. Inf. Syst. Secur.*, 9(3):259–291, August 2006.

[30] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46 –57, 31 1977-nov. 2 1977.

[31] Robert W. Reeder, Lujo Bauer, Lorrie Cranor, Michael K. Reiter, Kelli Bacon, Keisha How, and Heather Strong. Expandable grids for visualizing and authoring computer security policies. In *CHI 2008: Conference on Human Factors in Computing Systems*, pages 1473–1482, April 2008.

[32] Joe B Rhodes. Modular and distributive semilattices. *American Mathematical Society*, 201, 1975.

[33] W. Robinson. Monitoring software requirements using instrumented code. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, 2002.

[34] Nalin Saigal and Jay Ligatti. Defining and visualizing many-to-many relationships between concerns and code. Technical Report CSE-090608-SE, University of South Florida, September 2008.

[35] Thorsten Schäfer, Michael Eichberg, Michael Haupt, and Mira Mezini. The SEXTANT software exploration tool. *IEEE Transactions on Software Engineering*, 32(9):753–768, 2006.

[36] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *26th International Conference on Software Engineering (ICSE'04)*, pages 418–427, 2004.

[37] Henry Maurice Sheffer. A set of five independent postulates for boolean algebras, with application to logical constants. *Transactions of the American Mathematical Society*, 14(4):pp. 481–488, 1913.

[38] Macneil Shonle, Jonathan Neddenriep, and William Griswold. AspectBrowser for eclipse: A case study in plug-in retargeting. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, 2004.

[39] François Siewe, Antonio Cau, and Hussein Zedan. A compositional framework for access control policies enforcement. In *Proceedings of the 2003 ACM workshop on Formal methods in security engineering*, FMSE '03, pages 32–42, New York, NY, USA, 2003. ACM.

[40] D.A. Simovici and R.L. Tenney. *Theory of Formal Languages With Applications*. World Scientific, 1999.

[41] Wenjuan Xu, Mohamed Shehab, and Gail-Joon Ahn. Visualization based policy analysis: case study in selinux. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 165–174, New York, NY, USA, 2008. ACM.

APPENDICES

Appendix A Omitted Auxiliary Methods From Section 5.3

The `IncomingMail` specification omits the following method definitions, taken from [3]:

```
public static void log(Message m) {
    logFile = new PrintStream(new BufferedOutputStream(
                    new FileOutputStream("/examples/mail/incoming.log", true)));
    Message[] ma = GetMail.convertResult(m);
    if(ma==null) return;
    //log each message if it hasn't already been logged
    for(int i = 0; i<ma.length; i++) {
        String id = "";
        try {
        id = ma[i].getSentDate()+ma[i].getFrom()[0].toString()+ma[i].getSubject();
        } catch(Exception e) { }
        if(id.equals("")==false && loggedTable.containsKey(id)==false) {
            loggedTable.put(id, new Object());
            try {
                Enumeration e = ma[i].getAllHeaders();
                logFile.println("- - - - - <PoCo NEXT MESSAGE> - - --");
                while(e.hasMoreElements()) {
                    Header h = ((Header)(e.nextElement()));
                    logFile.println(h.getName() + ": " + h.getValue());
                }
                Object content = ma[i].getContent();
                if(content instanceof String)
                    logFile.println(content);
                else if(content instanceof Multipart) {
                    Multipart mmp = (Multipart)content;
                    for(int j = 0; j<mmp.getCount(); j++) {
                        BodyPart mbp = (BodyPart)(mmp.getBodyPart(j));
                        Object cont = mbp.getContent();
                        logFile.println("<PoCo>Multipart message, PART " + j + ":");
                        if(cont instanceof String) {
                            logFile.println("<PoCo:Type is " + mbp.getContentType() + ">");
                            logFile.println(cont);
                        }
                        else
                            logFile.println("<PoCo:Part not displayed; type is "
                            + mbp.getContentType() + ">");
                        logFile.println("");
                    }
                }
                else logFile.println("<PoCo>Unknown message type:\n" + content);
            }
        catch(Exception exn) {
            logFile.println("<PoCo>There was an error opening the mail:\n" + exn);
        }
        logFile.flush();
    }
}
```

```
public static String spamifySubject(Message m) {
    int MAX_SUBJ_LEN=32;
    String subj = m.getSubject();
    if(subj.length>MAX_SUBJ_LEN)
        subj=subj.substring(0, MAX_SUBJ_LEN);
    if(isSenderKnown(m)==false) {
        subj + "SPAM?: " + s;
        if(hasAttachment(m)) {
            JOptionPane.showMessageDialog(null,"This message contains an attachment that, if opened," +
            " could seriously harm\nyour computer. Unless you specifically asked the sender for" +
            " this attachment,\nit is strongly recommended that you delete this message immediately.",
            "Beware", JOptionPane.WARNING_MESSAGE);
        }
    }
    return subj;
}
```

Appendix A (Continued)

The `OutgoingMail` specification omits the following method definitions, taken from [3]:

```
public static void log(Message m) {
    try {
        logFile = new PrintStream(new BufferedOutputStream(
                        new FileOutputStream("/examples/mail/outgoing.log", true)));
        Enumeration e = m.getAllHeaders();
        logFile.println("- - - - - - - - - - <POLYMER NEXT MESSAGE> - - - - - - - - - -");
        while(e.hasMoreElements()) {
            Header h = ((Header)(e.nextElement()));
            logFile.println(h.getName() + ": " + h.getValue());
        }
        Object content = m.getContent();
        if(content instanceof String)
            logFile.println(content);
        else if(content instanceof Multipart) {
            Multipart mmp = (Multipart)content;
            for(int j = 0; j<mmp.getCount(); j++) {
                BodyPart mbp = (BodyPart)(mmp.getBodyPart(j));
                Object cont = mbp.getContent();
                logFile.println("<PoCo>Multipart message, PART " + j + ":");
                if(cont instanceof String) {
                    logFile.println("<PoCo:Type is " + mbp.getContentType() + ">");
                    logFile.println(cont);
                }
                else
                    logFile.println("<PoCo:Part not displayed; type is " + mbp.getContentType() + ">");
                logFile.println("");
            }
        }
        else logFile.println("<PoCo>Unknown message type:\n" + content);
    }
    catch(Exception exn) {
        logFile.println("<PoCo>There was an error opening the mail:\n" + exn);
    }
}
```

```
public static Integer confirm(Message m) {
    String confirmMsg = "You are sending email with subject\n";
    try {
        String subj = m.getSubject();
        if(subj.length() > 32)
            confirmMsg += " " + subj.substring(0, 30) + "..\nto the following address(es).\n";
        else
            confirmMsg += " " + subj + "\nto the following address(es)\n";
        Address[] recips = m.getAllRecipients();
        if(recips==null || recips.length==0)
            confirmMsg += "<could not find any recipients of message!>\n";
        else {
            for(int j = 0; j<recips.length && j<20; j++) {
                confirmMsg += " " + recips[j].toString() + "\n";
            }
            if(recips.length >= 20) confirmMsg += " ...\n";
        }
    }
    catch(Exception e) {
        confirmMsg += " <could not obtain subject/recipient of message>\n";
    }
    confirmMsg += "Select:\n "Yes" to allow this mail to be sent,\n";
    confirmMsg += " "No" to halt the email client (without sending this mail), or\n";
    confirmMsg += " "Cancel" to not send the currently outgoing mail but allow\n" +
                  " the email client to continue running.";
    return JOptionPane.showConfirmDialog(null, confirmMsg, "Security Question",
                    JOptionPane.YES_NO_CANCEL_OPTION));
}
```