3-20-2012

A Principled Approach to Policy Composition for Runtime Enforcement Mechanisms

Zachary Negual Carter
University of South Florida, zcarter@mail.usf.edu

A Principled Approach to Policy Composition for Runtime Enforcement Mechanisms

by

Zachary Carter

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Hao Zheng, Ph.D.
Les Piegl, Ph.D.

Date of Approval:
March 20, 2012

Keywords: Security monitor, policy specification, lattice theory, logic, Stirling numbers

ACKNOWLEDGEMENTS

I would like to acknowledge my major professor, Jay Ligatti, for his guidance and support, which made this work possible. Also, Daniel Lomsak, for without whom this work would also not be possible. I would also like to thank my friends and colleagues for their assistance, motivation, and humor that kept me sane.

TABLE OF CONTENTS

LIST OF TABLES iii

LIST OF FIGURES v

ABSTRACT vi

CHAPTER 1 INTRODUCTION 1
- 1.1 Related Work 1
- 1.2 Contributions 3
- 1.3 Organization of Thesis 4

CHAPTER 2 STATIC DECISION COMBINATORS 5
- 2.1 Definition 5
- 2.2 Static Committee Combinators 8
- 2.3 Specifying SDCs 9
  - 2.3.1 Combinators as Tables 10
  - 2.3.2 Combinators as Formulae 10

CHAPTER 3 EXAMPLES 12
- 3.1 Access Control 12
  - 3.1.1 Combinator Definition 13
  - 3.1.2 Combinator Decomposition 14
- 3.2 Quorums and Majority 16
  - 3.2.1 The Quorum Combinator 16
  - 3.2.2 The Majority Combinator 18
- 3.3 More General Purpose SCCs 20
  - 3.3.1 If-then-else Combinator 20
  - 3.3.2 Try-with Combinator 21
    - 3.3.2.1 Alternate Definition 21
  - 3.3.3 Dominates Combinator 22

CHAPTER 4 ALGEBRAIC PROPERTIES 23
- 4.1 Lattice Structure 23
  - 4.1.1 Partial Ordering of Votes 23
  - 4.1.2 Lattice Axioms 24

4.1.3 Distributivity 27
4.1.3.1 Distributive Elements 28
4.2 Negation 28
4.3 Automated Theorem Proving 29
4.3.1 Prover9 and Mace4 29
4.3.1.1 Input Format 30
4.3.1.2 Experience and Results 32
4.3.2 TPTP and E Prover 34

CHAPTER 5 ANALYSIS OF COMBINATOR SIZE 35

CHAPTER 6 NUMBER OF POSSIBLE COMBINATORS 42
6.1 Counting Combinators 42
6.2 Calculating the Number of SCCs 47

CHAPTER 7 CONCLUSION 49
7.1 Summary 49
7.2 Extensions 50
7.2.1 Tagged CounterProposals 50
7.2.2 Combination Algebra 50

LIST OF REFERENCES 52

APPENDICES 54
Appendix A SCC Definitions 55
Appendix B Prover9 and Mace4 Inputs and Proofs 58

LIST OF TABLES

Table 2.1	A partial table representation of an if-then-else-style combinator that illustrates details about how the permutations of subpolicy votes are listed.	11
Table 3.1	The abridged truth table for the ACP combinator with the three policies defined above.	13
Table 3.2	The complete definition of a conjunction SCC that favors the more restrictive operand.	15
Table 3.3	In the decomposed ACP, an intermediary result is computed between p_1 and p_2 before finding the final result by conjoining it with p_3.	15
Table 3.4	Inequality and disjunction SCCs utilized by the quorum combinator.	17
Table 3.5	In the table, the v variable represents any vote in V.	22
Table 4.1	Each element has a dual element, except for `CounterProposals`.	29
Table 5.1	Input permutations for an SDC that only takes counter proposals.	37
Table 5.2	Restricted partitions of a set of four elements with corresponding RGSs, grouped by the number of subsets.	39
Table 5.3	The table shows the mapping between SDC input permutations and the highlighted portion of the RGSs.	40
Table 6.1	The 16 boolean operators with an arity of 2.	42
Table 6.2	A 2-ary combinator with a domain and codomain of $\{Y, N, CP_1, CP_2\}$ has 10 possible variable assignments, each corresponding to a row ($s = 10$).	43
Table 6.3	4-restricted Stirling numbers of the second kind [1] appear as exponents in instances of equation 6.3.	47
Table 6.4	The number of SCCs increases at a much greater rate with arity than boolean or 4-valued logic functions.	48

Table A.1 Standard conjunction. 55
Table A.2 Standard disjunction. 55
Table A.3 Equality. 55
Table A.4 Inequality. 56
Table A.5 if-then-else 57

LIST OF FIGURES

Figure 2.1	The policy tree of a combinator defined using the formula: $p_1 \wedge (p_2 \vee p_3)$.	11
Figure 3.1	Decomposed ACP that uses two binary conjunction SCCs.	14
Figure 3.2	High-level organization of a ternary quorum combinator.	16
Figure 3.3	Low level construction of a ternary quorum combinator.	17
Figure 3.4	Construction of a ternary majority combinator.	18
Figure 3.5	A complete ternary majority-quorum SCC.	19
Figure 4.1	A partial ordering of SCC values based on restrictiveness.	24
Figure 4.2	A distributive lattice cannot have a diamond and pentagon as a sub-lattice.	28

ABSTRACT

Runtime enforcement mechanisms are an important and well-employed method for ensuring an execution only exhibits acceptable behavior, as dictated by a security policy. Wherever interaction occurs between two or more parties that do not completely trust each other, it is most often the case that a runtime enforcement mechanism is between them in some form, monitoring the exchange. Considering the ubiquity of such scenarios in the computing world, there has been an increased effort to build formal models of runtime monitors that closely capture their capabilities so that their effectiveness can be analysed more precisely. While models have grown more faithful to their real-life counterparts, issues concerning complexity and manageability (a common concern for software engineers) of centralized policies remains to be fully addressed. The goal of this thesis is to provide a principled approach to policy construction that is modular, intuitive, and backed by formal methods.

This thesis introduces a class of policy combinators adequate for use with runtime enforcement policies and analyses a particular instance of them called Static Committee Combinators (SCCs). SCCs present a model of policy composition where combinators act as committees that vote on events passing through the monitor. They were conceptualized in collaboration with Jay Ligatti and Daniel Lomsak. The general class of combinators are called Static Decision Combinators (SDCs), which share key features with SCCs such as allowing combinators to respond with alternative events when polled, in addition to responding with grants or denials. SDCs treat the base-level policies they compose as black

boxes, which helps decouple the system of combinators from the underlying policy model. The base policies could be modelled by automata but the combinators would not maintain their own state, being "static". This allows them to be easily defined and understood using truth tables, as well as analysed using logic tools. In addition to an analysis of SDCs and SCCs, we provide useful examples and a reusable combinator library.

CHAPTER 1

INTRODUCTION

Runtime enforcement mechanisms are found in most computer systems today, in one form or another. From browsers to operating systems, firewalls, and email clients— wherever there is interaction between systems a runtime enforcement mechanism is likely to be found monitoring the exchange. The implementation of a mechanisms also varies depending on the type of system it is defending. Sometimes it is favorable to in-line the monitor directly into an untrusted program, where in other cases it is preferable to have a middleman that intercepts events between systems. In each case, a system wishes to protect itself from potentially undesirable actions of an untrusted target and does so by having a security monitor enforce a policy that specifies acceptable and unacceptable behavior. When the monitor detects a breach in the policy, it may halt the interaction or try to transform it so that it becomes acceptable once more [10] [14].

1.1 Related Work

While monitors' ability to transform invalid traces into valid ones (such as when displaying a dialogue box to confirm an action that would otherwise be disallowed) is not new, only relatively recently have formal models been devised to capture policies with this ability. Two notable efforts are *edit automata* [10] and *Mandatory Results Automata* [11], among others [6]. These models define precisely what types of policies they can enforce, with each having their own advantages and disadvantages. They each provide a principled

approach to modelling runtime enforcement mechanisms and the policies they enforce, but we wish to pick up where they left off by providing a principled approach to composing said policies—one that is founded on *formal methods* [4].

Another proposed system to provide composability and modularity to runtime enforcement policies is the Polymer language [9]. Polymer is able to specify a rich set of policies and exhibits universal composability—meaning that decomposed policies retain all of the expressive power of a single monolithic policy. This further promotes the use of smaller, more modular *combinators* that can be reused. The design of Polymer and its specification of policies is geared toward real-world usage and as a result is less amendable to analysis. We sacrifice expresivity in our model to gain a simpler, more intuitive composition model.

For access-control policies, there have been a number of systems proposed for dealing with policy composition [7] [2] [3]. A recent and notable system is PBel, a policy-composition language that is capable of encoding conflicts and "gaps" in policy composition [3]. Conflicts arrise when there is a conjunction between two policies that disagree and gaps arrise when a request is irrelevant to a policy. PBel's basis in logic allows its policies to be statically analyzed with standard logic tools—a desirable property in a foundational system. PBel also treats the base building blocks (*request predicates*) of its policies as black boxes, which allows implementations to plug in different models without having to re-engineer a composition framework. PBel is, however, not viable for specifying general runtime enforcement policies since monitors should be able to insert arbitrary events into the trace when the policy requires it. Access-control policies only need to respond to requests with grants or denials. Also, encoding conflicts in a setting where there are more than two responses would not be possible with PBel's use of Belnap logic.

The system we propose to create policies for runtime enforcement mechanisms in a composable and modular manner addresses some of the concerns mentioned in these previous systems. Static Decision Combinators (SDCs), as they are called, provide the frame-

work for a generic composable layer over base policies, similar to operators in PBel, but allow policies and combinators to respond with arbitrary events—a necessary feature for facilitating policies that arbitrarily transform program traces. The model is limited enough that we can discover the entire space of possible combinators, while still allowing policies to give an unbounded number of different responses.

In this thesis, we focus on a particular instance of SDCs called Static Committee Combinators (SCCs), which carry the analogy of a committee in how they operate. The model was developed in collaboration with Jay Ligatti and Daniel Lomsak. SCCs vote with a `Yes`, `No`, `Abstain`, or are marked `NotPresent`. The committee tallies the votes and casts a decision. Further more, voters in the committees could be committees themselves, i.e. subcommittees. With SCCs, the problem of distributed resources that may prevent a proper vote is also addressed, with the `NotPresent` vote. We show how these combinators can be useful through examples and analysis.

1.2 Contributions

This paper introduces and analysis a system for composing policies for runtime enforcement mechanisms in Static Committee Combinators (SCCs), which are a part of a general class of combinators, Static Decision Combinators (SDCs). Specifically, our contributions are:

- A model for composing results given by runtime enforcement mechanisms called, SCCs, and a general class of SDCs.
- A small library of general-purpose and reusable SCCs.
- An analysis of the algebraic properties of SCCs and their use with automated theorem provers.

- An analysis of SCC and SDC complexity and how many distinct combinators can be defined.

1.3 Organization of Thesis

The remainder of this thesis is organized as follows: Chapter 2 describes and formally defines SDCs and SCCs. Chapter 3 gives various examples of SDCs and their applications. Chapter 4 explains the algebraic properties of a system where conjunctive and disjunctive SCCs form a lattice from votes. Chapter 5 analyzes how the size of SCC truth tables changes with arity and Chapter 6 uses that result to calculate just how many distinct SCCs exist. We wrap things up in Chapter 7 and illustrate promising extensions to SDCs and SCCs.

CHAPTER 2

STATIC DECISION COMBINATORS

This paper focuses on policy combinators—higher-order policies that combine the behaviors of other policies—that can be defined using a table or symbolically using formulae. We henceforth refer to them as Static Decision Combinators (SDCs) and in a particular case, Static Committee Combinators (SCCs). We study these combinators because they can express many common combination strategies (e.g. conjunctive combinators), they are amenable to formal analysis due to their static nature, and the investigation of this combinator class may provide a foundation that can be built upon in order to study more expressive classes.

2.1 Definition

SDCs are defined within the context of an environment with a set of events E. A sequence of events from E flow between two entities through a security monitor. The monitor should at least be able to forward an event, insert a new event, or halt the flow of events. These decisions form the basis of responses that policies and combinators give. Policies respond `Yes` to indicate the event should be forwarded and `No` to indicate that the monitor should halt. Policies can also respond with a `CounterProposal(e)` to indicate a different event e should be inserted into the sequence. We will often refer to these responses as *votes*, which reinforces the analogy used later to describe SCCs. The

environment includes the set of possible votes as well, with the baseline we've established $V = \{Yes, No\} \cup CounterProposal(E)$ as well as any additional "constant" votes.

Definition 1 *An* environment *consists of a possibly countably infinite set of events E and a set of votes V, $V = \{Yes, No\} \cup CounterProposal(E) \cup C$, where C is a set of additional constant votes. $CounterProposal(E)$ represents the set of counterproposals, where each counterproposal vote has a corresponding event from the set E.*

SDCs are similar to digital circuits in combinatorial logic in that they are pure functions; their output is dependent only on the inputs given at a particular moment and not on past inputs. Because they lack memory, adding events to the trace does not affect their behavior. They are also general-purpose in that they cannot make decisions based on the content of a counterproposal. At most, they can discern whether any two proposed events are equal to each other. In this sense, we say that `CounterProposal` votes are *opaque*. Additionally, for each event they are to vote on, they consult each subpolicy on that same event and exactly once.

Another distinguishing feature of SDCs are the votes they are capable of producing. In general, the votes include a number of *constants* such as `Yes`, `No`, or others, and an unbounded number of `CounterProposals` that are comprised of system events E. The domain and co-domain of SDCs is then the union of the set of constants C and the system event set E, which we signify as V (for "votes").

How the votes are ultimately interpreted by the system is up to the implementation, though when receiving a vote of `Yes` it should be expected that the system executes the event being voted on, and that upon receiving a `CounterProposal` the proposed event should be executed. With SCCs we will see other types of votes with their own expected interpretations. Whatever interpretations are chosen at the system level, it will not affect

the behavior of combinators, which simply use the votes from subpolicies to decide on what vote to cast.

Definition 2 *A* SDC *is a higher-order policy that takes n subpolicies and produces a composite policy with a particular voting function: $V_1 \times \cdots \times V_n \to V$*

The voting function is what characterizes a particular SDC, and can be defined using a truth table. An example is provided in the following section on SCCs.

Combinators are composed with subpolicies that themselves could be combinators, building up a tree recursively until the leaf nodes, which are *base policies*.

Definition 3 Base policies *are treated as black boxes that have a query function with a domain of the event set E and a co-domain of the vote set $E \cup C$, where C is the set of constants.*

Definition 4 *A* policy tree *is defined recursively as either an SDC with policy trees as subpolicies or a base policy. Policy trees show the internal structure of a composite policy, unless they are a single base policy.*

A base policy can appear multiple times within a policy tree, and when it does it is assumed that each instance will give an identical vote. Depending on the implementation, this could mean the policy is queried once and its vote is substituted wherever the policy appears in the tree, or alternatively that the query function is idempotent throughout the query process of the tree. We remain as abstract as possible on how the internal workings of querying works, so long as SDCs remain pure functions operating on votes from their specified subpolicies.

For each input event e that would be appended to the execution, the policy being enforced casts a vote that indicates how the trace should be affected. This vote is the final result from the combinator at the base of the tree.

It should also be noted that while composite policies may exhibit this tree structure, mechanisms need not be separate or distributed. The policies, while possibly defined by distributed parties, could be collected and enforced in a single mechanism, such as an Inline Reference Monitor [5] or Program Monitor [9].

2.2 Static Committee Combinators

Static Committee Combinators (SCCs) are a particular flavour of SDCs where combined policies are viewed as committees that vote on proposed system events. A combinator acts as the chair who collects the votes of the members (who may in turn chair their own sub-committee) and casts the ultimate vote. This is not unlike SDCs in general where the combinator is the decider, but for SCCs the set of votes and behaviors carry the committee analogy. The complete set of votes V for SCCs is $\{Y, N, A, CP(E), NP\}$ described as follows:

- `Yes` (Y): e should be output.
- `No` (N): e should not be output. If `No` is the ultimate vote, execution freezes at this point because there is no event to be added to the trace.
- `Abstain` (A): The policy neither opposes nor endorses e. This cannot be the ultimate vote because `Abstain` does not suggest any way to affect the execution. Policy designers can use a root policy to interpret `Abstain` at the topmost level (i.e., a policy that transforms `Abstain` into some other vote e.g. `Yes` or `No`). `Abstain` could be used to indicate conditions such as a policy's lack of interest in (or knowledge of) e, the absence of a clear majority vote amongst a combinator's children, etc.

- `CounterProposal`(e) (CP): Alternative event e' should be output instead of e. The ability to emit `CounterProposal`s of arbitrary events are what make these policies expressive because, unlike truncation-based models, policies can require executions to be modified (i.e., they can edit executions like MRAs).

- `NotPresent` (NP): the policy is unavailable or lacks some critical resource necessary for it to vote. `NotPresent` allows policies to express exceptional circumstances and could be emitted by a policy that requires access to an unavailable resource or used to signify a communication failure when the elements of the policy tree are distributed across a network. If `NotPresent` is the ultimate vote, then execution halts (in practice, `NotPresent` might be an exception that is thrown rather than a value that is returned). Since policies are not required to be decidable, they may diverge, which is also represented by `NotPresent`. In practice it would be difficult to determine for what reason a policy a policy is unreachable. Following the committee analogy, a divergent policy could be seen as a filibuster.

We believe that the above list of outcomes is complete. Each voter is either present or not (`NotPresent`). If present, each voter can vote or not. A voter can not vote by explicitly refusing (`Abstain`) or by never casting a vote ($\perp$). Voters can vote yes, no, or propose a completely different event to output.

2.3 Specifying SDCs

As functions, combinators could be defined using standard mathematical notation. For instance, an if-then-else-style combinator could be defined as a function:

$$A(p_1, p_2, p_3) = \begin{cases} p_2 & \text{if } p_1 = Y \\ p_3 & otherwise \end{cases}$$

The behavior is clear from the definition, though as with functions of boolean or multi-valued logic, it can be helpful to define them using truth tables and formulae.

2.3.1 Combinators as Tables

All SDCs can be formulated as truth tables in which the output of the SDC is determined by the outputs of its subpolicies. The ability of committee policies to output counterproposals complicates the representation and reasoning about them compared to functions in a straightforward multi-valued logic. This complication arises because a system may have infinitely many events and each counterproposal is parametrized over an event. Therefore, there are potentially an infinite number of unique counterproposals that a policy can vote with.

Infinitely large tables are not required because SDCs do not have any knowledge of particular events and so counterproposed events can be represented abstractly using event variables. However, SDCs can determine if two events are equal without any special knowledge. Thus, in cases where more than one subpolicy outputs a counterproposal, an SDCs can cast a different vote for each combination of equality relationships amongst the counterproposed events. Event variables are used to label counterproposals such that the equivalence relationships amongst the variable identifiers signifies the equivalence relationships amongst the events they represent. Table 2.1 illustrates the definition of a ternary combinator called ITE (if-then-else) that uses p_1 as the predicate such that ITE votes like p_2 when p_1 votes Y and otherwise votes like p_3.

2.3.2 Combinators as Formulae

Combinators can be defined as a function or in terms of other combinators with operator symbols.

Table 2.1. A partial table representation of an if-then-else-style combinator that illustrates details about how the permutations of subpolicy votes are listed.

p_1	p_2	p_3	$ITE(p_1, p_2, p_3)$	Comments
N	Y	A	A	p_1 does not vote Y, so ITE votes the same as p_3
Y	NP	CP_1	NP	p_1 votes Y, so ITE votes the same as p_2
CP_1	CP_1	CP_1	CP_1	p_1 is not Y, so vote p_3. All subpolicies vote the same counterproposal.
CP_1	CP_2	CP_1	CP_1	p_1 is not Y, so vote p_3. p_1 and p_3 vote the same counterproposal; p_2, a distinct counterproposal.
CP_1	CP_2	CP_3	CP_3	p_1 is not Y, so vote p_3. All subpolicies vote counterproposals, each of them distinct events.
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

If combinators have been defined and given operator symbols, we can define new combinators in terms of those symbols. For example, say we have conjunction and disjunction combinators with operators $\wedge$ and $\vee$ respectively. We can define a new ternary combinator as $C = p_1 \wedge (p_2 \vee p_3)$. A visual representation of this combinator would, infact, be a policy tree.

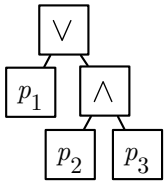

Figure 2.1. The policy tree of a combinator defined using the formula: $p_1 \wedge (p_2 \vee p_3)$.

In order to verify the behavior of the formula one could write out the results, including sub-formulae, in a truth table as is often seen in boolean logic. We will see more examples of this in Chapter 3.

CHAPTER 3

EXAMPLES

This chapter demonstrates different types of policies SCCs can implement and how they can be constructed in a modular fashion.

3.1 Access Control

We can imagine a scenario where there are two users on a system, Alice and Bob, who have access to a resource, R, but only one user may access the resource at a time. Instead of specifying the requirements as a single policy, we can create a composite of three policies using a ternary SCC. We will refer to this policy as *ACP* for Access-Control Policy.

The first policy decides whether Alice is allowed to access resource R. If the action being voted on is in fact Alice accessing resource R, the policy returns `Yes`. If it is some other action, the policy returns `abstain`. The second policy is for Bob and behaves analogously, allowing Bob to access R or otherwise abstaining.

The third policy specifies that only 1 user is allowed to access resource R. If no user is currently accessing R and either Alice or Bob attempts to, the policy will vote `Yes`. However, if a user is currently accessing R when another attempt is made, the policy will vote `No`. If the action is not an access on R, the policy will abstain from voting. This policy will use some means to maintain its security state, though this is outside the concern of the combinators.

It is conceivable that the third policy could propose some other action to take if the resource is busy by returning a `CounterProposal`. The SCC could account for this and return the `CounterProposal` instead of `No`, but for the purposes of demonstrating access control this example sticks with `No`.

3.1.1 Combinator Definition

The three subpolicies p_1, p_2, and p_3 are combined using a ternary SCC that produces the desired policy specification. The composite policy should abstain when all three subpolicies abstain, and permit the action when either p_1 or p_2 return `Yes` and p_3 returns `Yes`. The policy should otherwise return `No` to indicate the access has failed. A truth table description of the SCC shows exactly which inputs provide which output.

Table 3.1. The abridged truth table for the ACP combinator with the three policies defined above. It shows outcomes where Alice or Bob may not have access to the resource in addition to our hypothesized scenario where they do. We also show cases where a policy returns NP, indicating some failure occurred preventing the subpolicy from formulating a proper vote. Inputs that cannot occur, such as p_1 being CP_1, are omitted from the table to simplify. If p_1 did emit those values, the table would be incomplete without rows for those inputs.

p_1	p_2	p_3	$ACP(p_1, p_2, p_3)$
N	NP	Y	N
N	A	Y	N
NP	N	Y	N
NP	NP	Y	N
NP	Y	Y	Y
NP	A	Y	N
Y	NP	Y	Y
Y	A	Y	Y
A	N	Y	N
A	NP	Y	N
A	Y	Y	Y
A	A	A	A

The composite policy exhibits the desired behavior we expected, and we are able to modify the policy by simple switching one of its subpolicies with a different one without modifying the SCC. For example, we could change the policy to deny Bob access to the resource by changing p_2.

3.1.2 Combinator Decomposition

SCCs can often be decomposed into more modular and general SCCs that would likely be found in a combinator library for distribution and reuse. This particular access-control policy can be decomposed using only a conjunctive binary SCC (signified with the $\wedge$ operator). To emulate the behavior of the ternary SCC, the conjunction can operate on p_1 and p_2 to determine whether one of them permits the action or whether both abstain, then another can operate on the result of the first and whether the resource is available (the vote of p_3).

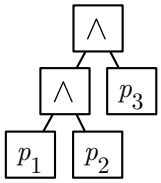

Figure 3.1. Decomposed ACP that uses two binary conjunction SCCs.

Intuitively it may seem the conjunction between p_1 and p_2 should rather be a disjunction, asking whether Alice *or* Bob is permitted to access the resource. Conjunction works in this instance because only the policy concerned with the user initiating the action will vote, rather than abstain, and a typical interpretation of policy conjunction is to take the most restrictive result, and a vote (*any* vote) is more restrictive than abstention. If a policy abstains from voting and another produces a vote, it is reasonable to expect the conjunction to follow the policy that voted, which achieves the desired behavior of voting `Yes` whenever p_1 or p_2 vote `Yes`. If both abstain, their conjunction will also abstain.

The second conjunction operates on the result of the first conjunction (which determined if the user was authorized to access the resource) and p_3 (which determines if the resource is still available). Since our definition of conjunction returns the most restrictive vote, and `No` is more restrictive than `Yes`, it is straightforward to show that the composed policy will deny the access if p_3 returns `NotPresent` and permit it if p_3 returns `Yes`, just as the original ternary SCC would. If p_3 abstains, the action was not a resource access and the other subpolicies will have abstained also, resulting in each conjunction and the composite policy producing an abstain.

Table 3.2. The complete definition of a conjunction SCC that favors the more restrictive operand.

$\wedge$	N	NP	CP_1	CP_2	Y	A
N	N	N	N	N	N	N
NP	N	NP	NP	NP	NP	NP
CP_1	N	NP	CP_1	NP	CP_1	CP_1
Y	N	NP	CP_1	CP_2	Y	Y
A	N	NP	CP_1	CP_2	Y	A

We have shown that it is possible to build a policy tree with the same behavior as a custom designed ternary SCC by using a generic conjunction SCC, increasing the modularity of policy composition and reuse of combinator parts.

Table 3.3. In the decomposed ACP, an intermediary result is computed between p_1 and p_2 before finding the final result by conjoining it with p_3. Notice that the results in the final column match the entries of the final column in Table 3.1 of the ACP combinator.

p_1	p_2	p_3	$p_1 \wedge p_2$	$(p_1 \wedge p_2) \wedge p_3$
N	A	N	N	N
N	A	Y	N	N
Y	A	N	Y	N
Y	A	Y	Y	Y
A	N	N	N	N
A	N	Y	N	N
A	Y	N	Y	N
A	Y	Y	Y	Y
A	A	A	A	A

3.2 Quorums and Majority

SCCs can also be used to build policies that return arbitrary actions to the system.

3.2.1 The Quorum Combinator

The next example is of an SCC that enforces a $quorum$, a relevant concept for committees engaged in decision making. If the quorum (the minimum number of participants that must be present to vote) is not met, the SCC votes `No`. At a high level, there is a conjunction between an SCC that checks if the quorum has been met, returning `No` if the quorum has not been met and `abstain` if it has, and the original SCC, which is composed of a number subpolicies.

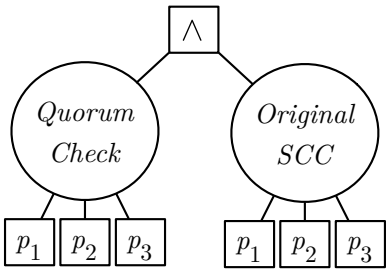

Figure 3.2. High-level organization of a ternary quorum combinator.

At a lower level the quorum combinator checks each quorum-sized combination of subpolicies to make sure that at least one such combination has no subpolicies returning `NotPresent`. For example, if a policy has three subpolicies and the quorum is set to 2, the combinations of subpolicies would be $\{(p_1, p_2), (p_1, p_3), (p_2, p_3)\}$. If none of these combinations have all subpolicies present, then the quorum has not been met. In order to check the votes of subpolicies, we can define a new inequality SCC, symbolized with $\neq$. In addition to inequality, we define a disjunction SCC ($\vee$) that can be used to ensure that at least one combination meets the quorum.

Table 3.4. Inequality and disjunction SCCs utilized by the quorum combinator. The inequality combinator is used to check that subpolicies' votes are not equal to `NotPresent`, and the disjunction SCC is used to ensure that at least one combination of subpolicies meets the quorum.

$\neq$	N	NP	CP_1	CP_2	Y	A
N	N	A	A	A	A	A
NP	A	N	A	A	A	A
CP_1	A	A	N	A	A	A
Y	A	A	A	A	N	A
A	A	A	A	A	A	N

$\vee$	N	NP	CP_1	CP_2	Y	A
N	N	NP	CP_1	CP_2	Y	A
NP	NP	NP	CP_1	CP_2	Y	A
CP_1	CP_1	CP_1	CP_1	Y	Y	A
Y	Y	Y	Y	Y	Y	A
A	A	A	A	A	A	A

The checks of each quorum-sized combination are merged together (using disjunction) into a policy tree that returns `Abstain` if at least one of the combinations of subpolicies meet the quorum and `No` otherwise.

The policy tree that forms the quorum-checking policy is then conjoined with the original SCC so that if the quorum is reached, the original vote is cast, or else N is cast. The exact construction of the quorum combinator depends on the arity of the original SCC and the value chosen for quorum, though the pattern should be simple enough generate the designed combinator using tools. Figure 3.3 displays a ternary SCC labeled "Original SCC" composed with a quorum-check that enforces a quorum of two.

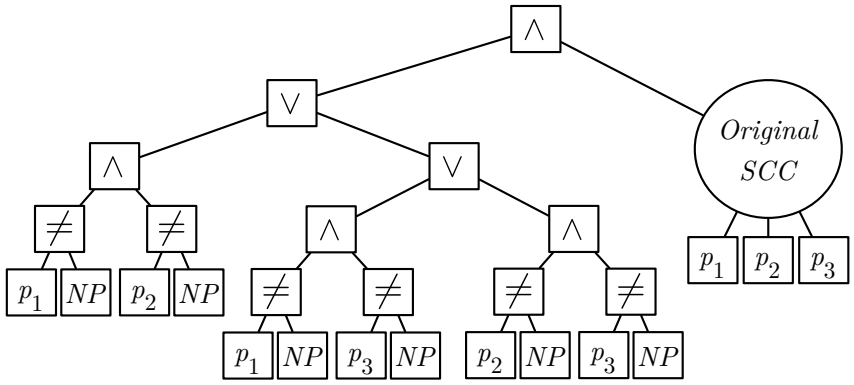

Figure 3.3. Low level construction of a ternary quorum combinator.

3.2.2 The Majority Combinator

Another useful type of SCC is one that always returns the majority vote. As with the quorum combinator, the low level construction of such a combinator will depend on its arity (n). A way to construct the majority combinator is to check each combination of subpolicies (where the number of subpolicies in each combination is equal to $\lceil n/2 \rceil$) and if the subpolicies cast the same vote, return that vote, or else vote `No`. For the ternary case, the combinations of subpolicies would be $\{(p_1, p_2), (p_1, p_3), (p_2, p_3)\}$, so we can check for a majority by seeing if any of these pairs gives the same value. To compare votes, we'll assume we have an equality SCC ($=$) that is the dual of $\neq$, returning `Abstain` when they are equal and `No` when they are not.

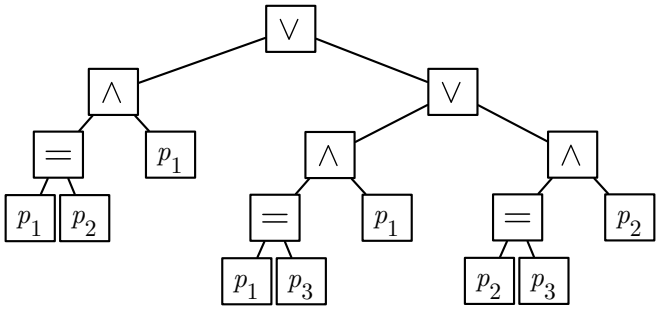

Figure 3.4. Construction of a ternary majority combinator.

Since $=$ will return `Abstain` if both operands are equal and `Abstain` conjoined with any other vote gives that other vote, the conjunction $(p_1 = p_2) \wedge p_1$ will return p_1 when p_1 casts the same vote as p_2, which will give the majority vote. If the second operand of the conjunction was p_2 it would work the same, since all subpolicies would have the same value if they're equal. The choice of which policy's vote will be returned is arbitrary. If the equality check fails for each combination, the disjunction of them all will return N.

For arities greater than 3, the complexity of the low-level construction increases greatly. With an arity of 4, for instance, a majority could be a combination of 2 or 3 subpolicies,

so combinations of each size would need to be checked – first for majorities of 3, then for 2. For combinations that are less than or equal to half the arity in size, such as 2 when the arity is 4, there could be ties for majority. A special disjunction may need to be created to deal with ties. Alternatively, a custom n-ary SCC could be created instead of trying to decompose it into generic SCCs.

The majority combinator has a potential undesirable behavior when the majority of subpolicies return `NotPresent`. If `NotPresent` is considered an undesirable majority vote, the policy author could use a Quorum Combinator to ensure there are enough subpolicies present to reach a majority. If there are not enough subpolicies for a majority, then it votes `No`. Figure 3.5 shows how the quorum and majority combinators can be combined, with the majority combinator replacing the place holder policy labelled "Original SCC" in the figure showing the quorum combinator, Figure 3.3.

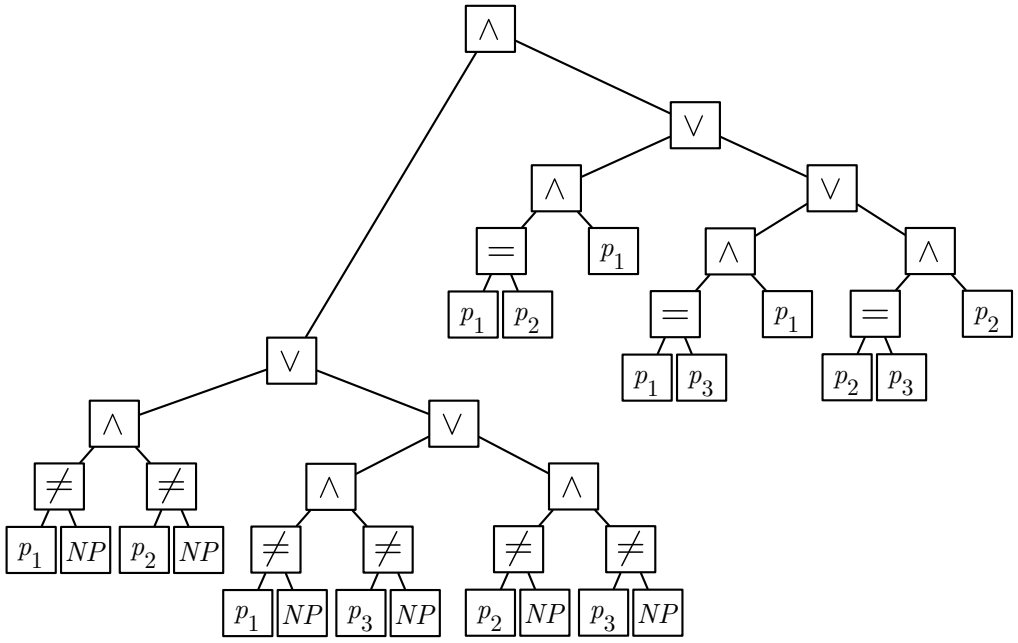

Figure 3.5. A complete ternary majority-quorum SCC.

This combinator will produce the majority vote of its subpolicies, unless there was no majority or the quorum of 2 was not met.

3.3 More General Purpose SCCs

The conjunction, disjunction, equality, and inequality combinators that have been introduce in previous examples (Tables 3.2 and 3.4) can be used to create additional general purpose combinators.

3.3.1 If-then-else Combinator

Another useful, general purpose SCC is the *if-then-else* combinator. This combinator is ternary: the first subpolicy acts as a condition, which, if it votes `Abstain` or `Yes`, means the combinator will cast the vote of its second subpolicy, otherwise it will cast the vote of its third subpolicy (full definition in Figure A.5). The policy can be defined in terms of $\vee$, $\wedge$, and $=$:

$$ITE(p_1, p_2, p_3) = ((p_1 = A \vee p_1 = Y) \wedge p_2) \vee ((p_1 \neq A \wedge p_1 \neq Y) \wedge p_3)$$

Key to the decomposition of this combinator is that at least one side of the disjunction will always be `No`, which means the vote from the other side will be favored in a disjunction. This is because $(p_1 = A \vee p_1 = Y) \wedge p_2)$ will produce an `Abstain` precisely when $(p_1 \neq A \vee p_1 \neq Y)$ produces a `No`, and vice-versa. When the dual equality checks are conjoined with their respective subpolicies, one will produce the vote of the policy and the other will produce a `No` (recall that conjunction with `Abstain` returns the other operand, and conjunction with `No` returns `No`).

By reusing our basic combinators we are able to achieve the desired behavior of an if-then-else combinator, which could be used to define other generic combinators. For example, we could define a binary *try-with* combinator as $TRY(p_1, p_2) = ITE(p_1 = Y, p_1, p_2)$.

3.3.2 Try-with Combinator

This combinator votes `Yes` if the first policy does, otherwise it casts the vote of p_2. Of note in the definition $TRY(p_1, p_2) = ITE(p_1 = Y, p_1, p_2)$ is how a formula was supplied as an argument to the ITE combinator, which works because formulae represent composite policies. This nested approach can be used to express chains of combinations, forming higher-arity policy trees. For example, here is a series of nested try-withs that form a 5-ary policy:

$$TRY(p_1, TRY(p_2, TRY(p_3, TRY(p_4, p_5))))$$

This chain of try-withs forms a policy that will cast `Yes` if any of the first four policies do, or else it will cast the vote of the last policy, p_5. This technique could be used with if-then-else as well for even more intricate behavior.

3.3.2.1 Alternate Definition

The previous definition of try-with using the ITE combinator would fully expand to:

$$(((p_1 = Y) = A \vee (p_1 = Y) = Y) \wedge p_1) \vee (((p_1 = Y) \neq A \wedge (p_1 = Y) \neq Y) \wedge p_2)$$

Some parts of the formula are extraneous— unessential to achieving the behavior of try-with. A more direct, simplified version would be:

$$TRY(p_1, p_2) = ((p_1 = Y) \wedge p_1) \vee ((p_1 \neq Y) \wedge p_2)$$

The formula can be decomposed and each part listed in a truth table to show the behavior of the formula still matches the desired behavior. The decomposition is given in Table 3.5.

Table 3.5. In the table, the v variable represents any vote in V. The try-with formula has a disjunction of the two sub-formulae in columns 3 and 5. By inspecting the columns, we can see there is either a disjunction of $N \vee v$, which always gives v, and $Y \vee N$, which gives Y, and that Y is only the result when $p_1 = Y$ and in all other cases it is v, giving us the desired behavior.

p_1	p_2	$p_1 = Y$	$(p_1 = Y) \wedge p_1$	$p_1 \neq Y$	$(p_1 \neq Y) \wedge p_2$	$TRY(p_1, p_2)$
N	v	N	N	A	v	v
NP	v	N	N	A	v	v
CP_1	v	N	N	A	v	v
Y	v	A	Y	N	N	Y
A	v	N	N	A	v	v

3.3.3 Dominates Combinator

This SCC should return whatever value the first subpolicy proposes, otherwise it will return the value of the second subpolicy. In this case, there are two votes we will consider "non-votes": `Abstain` and `NotPresent`. In such cases, we will defer the decision of what to vote to the second subpolicy. We can formulate the policy as:

$$DOM(p_1, p_2) = ((p_1 \neq A \vee p_1 \neq NP) \wedge p_1) \vee ((p_1 = A \vee p_1 = NP) \wedge p_2)$$

This policy uses the same technique as if-then-else and try-with, using a disjunction of a comparison and its negation, so that only one side will produce a non `No` vote.

CHAPTER 4

ALGEBRAIC PROPERTIES

Combinators that exhibit certain algebraic properties are useful for policy writers when designing policy trees and also for performing static analysis using automated theorem provers.

4.1 Lattice Structure

Lattices are a basic algebraic structure that have a useful dual interpretation relating to order theory—both interpretations become useful for reasoning about votes and their combinations. In regards to order theory, we can consider some votes as potentially *more impactful* on the monitor's trace than others and establish an ordering relation among them based on observation. As an algebraic structure, a lattice includes operations with algebraic properties such as commutativity, associativity, and others, which are vital for analysis.

4.1.1 Partial Ordering of Votes

An ordering of votes based on their potential to impact the trace provides guidance on what their combinations should produce. Intuitively, a conjunction between a high impact vote and a low impact vote should favor the higher impact vote, while the opposite is true for disjunction. For SCCs, `Abstain` has the least impact since it will not affect the trace or influence the vote of another policy when conjoined. A vote of `Yes` has more potential for impact as it means the policy actively wishes to preserve the

trace. `CounterProposals` will alter the trace, so have move potential for impact than `Abstain` or `Yes`. `NotPresent` is less restrictive than `No` as it is not an outright denial, but is more restrictive than `Yes` and `Abstain` since it could have potentially been a denial. `CounterProposals` are in the middle, with a restrictiveness somewhere between granting and denying, or possibly denying. `CounterProposals` are also not comparable to each other in this ordering because thier semantics, and thus their restrictiveness, is opaque to the combinator.

Using the restrictiveness ordering, we can arrive at the lattice structure in Figure 4.1.

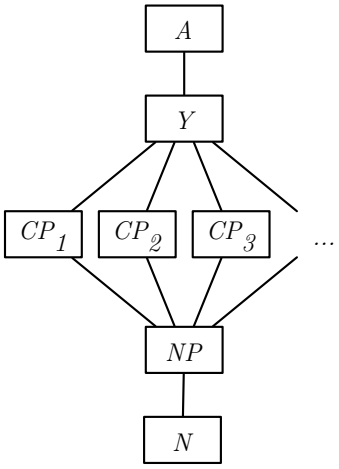

Figure 4.1. A partial ordering of SCC values based on restrictiveness.

4.1.2 Lattice Axioms

The conjunctive and disjunctive combinators introduced in the previous chapter (shown in Tables 3.2 and 3.4) are based on the same restrictiveness ordering. They form the *meet* and *join* operations, respectively, two operations that characterize lattices. Given a lattice structure, such as the one for SCCs $(\{A, Y, NP, N, CP_1, ...\}, \wedge, \vee)$, there are a number of

axiomatic identities that hold for conjunction and disjunction over all latice elements x, y, z:

$$x \vee y = y \vee x \qquad \text{(commutativity)}$$

$$x \wedge y = y \wedge x \qquad \text{(commutativity)}$$

$$x \vee (y \vee z) = (y \vee x) \vee z \qquad \text{(associativity)}$$

$$x \wedge (y \wedge z) = (y \wedge x) \wedge z \qquad \text{(associativity)}$$

$$x \vee (x \wedge y) = x \qquad \text{(absorption)}$$

$$x \wedge (x \vee y) = x \qquad \text{(absorption)}$$

$$x \vee x = x \qquad \text{(idempotency)}$$

$$x \wedge x = x \qquad \text{(idempotency)}$$

Idempotency can be derived from absorption, so it is not usually included as an axiom—we include it here because it is a notable property and the derivation is not immediately obvious. By setting $y = x \wedge z$, the absorption property becomes $x \wedge (x \vee (x \wedge z)) = x$, which simplifies to $x \wedge x = x$.

From these axioms, which all lattices share, one can define a less-than-or-equal relation, $\leq$, that gives the elements of the lattice a partial ordering:

$$x \leq y \iff x \wedge y = x \qquad \text{(less-than-or-equal)}$$

For convenience, we also define a less-than relation $<$:

$$x < y \iff x \leq y \; and \; x \neq y \qquad \text{(less-than)}$$

Additional axioms are necessary to produce the lattice of votes shown in Figure 4.1:

$$x \vee N = x \qquad \text{(join identity)}$$

$$x \wedge A = x \qquad \text{(meet identity)}$$

$$N < NP \qquad \text{(NP element)}$$

$$N < x \Rightarrow NP \leq x \qquad \text{(NP is atomic)}$$

$$Y < A \qquad \text{(Y element)}$$

$$x < A \Rightarrow x \leq Y \qquad \text{(Y is coatomic)}$$

$$CP(x) \iff \neg(x = Y \; or \; x = NP \; or \; x = A \; or \; x = N) \qquad \text{(counterproposal)}$$

$$(CP(x) \; and \; CP(y) \; and \; x \neq y) \Rightarrow (x \wedge y = NP \; and \; x \vee y = Y)$$

(counterproposal non-comparable)

The *join* and *meet identity* establish bounds on the lattice, with N being the least element and A being that greatest element. An *atomic element* is one that *covers* the least element, meaning that it is less than all other elements besides the least element. The coatomic element is the dual concept. The predicate $CP(x)$ indicates that votes are counterproposals if and only if they are not one of the constants. The *counterproposal non-comparable* axiom establishes the diamond shape in the middle of the lattice since the join and meet of any distinct counterproposals is always above and below.

From these properties we can see that conjunction and disjunction for SCCs work similar to how they do for Boolean and other logics. Commutativity for SCCs guarantees that

the order of subpolicies does not affect the outcome, while associativity means that, in a homogeneous tree of SCCs, the hierarchy also will not affect the outcome.

A noticeable property that SCCs lack is distributivity. We will see that this is common of SDCs based on a lattice structure.

4.1.3 Distributivity

Distributivity is a necessary property for a number of algebraic structures, namely groups, distributive lattices, and Boolean algebras. In order for distributivity to hold, the following laws must hold for all elements a, b, c:

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$
$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$$

The laws break down for SCCs when all elements are `CounterProposals`:

$$CP_1 \wedge (CP_2 \vee CP_3) = (CP_1 \wedge CP_2) \vee (CP_1 \wedge CP_3)$$
$$CP_1 \wedge Y = NP \vee NP$$
$$CP_1 = NP$$

In general, any lattice that contains a diamond or a pentagon shape (as shown in Figure 4.2 cannot be distributive ([16] Theorem 1.2.1), and a diamond shape can clearly be seen in Figure 4.1 between Y, NP, and the counterproposals. Any SDCs that use a similar ordering and have at least three distinct `CounterProposals`, enough to form a diamond, will also have this limitation.

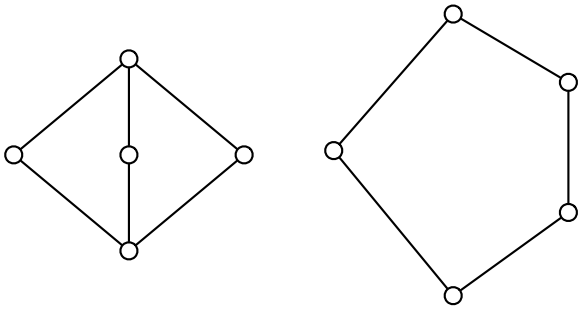

Figure 4.2. A distributive lattice cannot have a diamond and pentagon as a sublattice. A modular lattice cannot have a pentagon as a sublattice.

4.1.3.1 Distributive Elements

While general distributivity may not be possible, we can consider if certain elements have this useful property, which would allow policies to be written as if it were true if the conditions are right.

An element d is said to be distributive ([16] page 76) if $d \vee (x \wedge y) = (d \vee x) \wedge (d \vee y)$ holds for all x, y in the set of lattice elements.

For the SCC lattice, `CounterProposals` are the only non-distributive elements.

4.2 Negation

A negation combinator is useful for building composite policies such as $\neq$ from $=$. In the table for the negation SCC, every element has a dual except for `CounterProposals`, which return themselves. This shows that negations does not produce *complements* in general (an element x has a complement iff $x \vee \neg x = 1$ and $x \wedge \neg x = 0$).

Interestingly, even though De Morgan's law ($(x \wedge y) = \ x \vee \ y$) pertains to elements and their complements, the formula holds using our meet, join, and negation operations. A De Morgan algebra is defined as a bounded distributive lattice with a unary operator such that [8]:

Table 4.1. Each element has a dual element, except for `CounterProposals`.

p	$\neg p$
N	A
NP	Y
CP	CP
Y	NP
A	N

$$\neg(\neg x) = x$$

$$\neg(x \wedge y) = \neg x \vee \neg y$$

Though not technically a De Morgan algebra, our lattice and negation operator do satisfy the conditions. We provide proofs using an automated theorem prover.

4.3 Automated Theorem Proving

The algebraic properties of combinators based on the lattice structure are not only useful for human reasoning and understanding, it also allows them to be formally analysed using automated tools. This section describes initial experiences using automated theorem provers on SCCs.

4.3.1 Prover9 and Mace4

Prover9 and Mace4 are a duo of programs that come bundled together and operate on the same input. Prover9 is an automated theorem prover for first-order and equational logic, while Mace4 searches for finite models and counterexamples [12]. The user supplies a set of assumptions and a set of goals and either instructs Prover9 to find a proof or Mace4 to

find a counterexample. The input format is such that the lattice axioms can be used almost verbatim for the set of assumptions.

4.3.1.1 Input Format

Various options and declarations for custom operators are placed at the beginning of the file:

```
% Language Options
op(325, prefix, "~").
op(600, infix, "==").

if(Prover9). % Options for Prover9
  assign(max_seconds, 120).
end_if.

if(Mace4).   % Options for Mace4
  assign(max_seconds, 60).
end_if.
```

The custom operators declared at the top are necessary for specifying the negation and equality combinators later on. The `max_seconds` options allow the user to limit how long the programs will run before giving up.

Next after these options come the assumptions. First, we have the lattice axioms:

```
formulas(assumptions).
% Lattice Theory
x ^ y = y ^ x                # label("commutativity_meet").
x v y = y v x                # label("commutativity_join").
(x ^ y) ^ z = x ^ (y ^ z)    # label("associativity_meet").
(x v y) v z = x v (y v z)    # label("associativity_join").
(x v y) ^ x = x              # label("absorption_1").
```

```
(x ^ y) v x = x                  # label("absorption_2").
% define relations
x <= y <-> x ^ y = x             # label("less_than_equal").
x < y <-> x <= y & x != y        # label("less_than").
% SCC votes
x v N = x                        # label("join_identity").
x ^ A = x                        # label("meet_identity").
N < NP.
Y < A.
N < x -> NP <= x                 # label("atomic_element").
x < A -> x <= Y                  # label("coatomic_element").
% counterproposals
CP(x) <-> -(x = Y |
            x = NP |
            x = A | x = N)   # label("counter_proposal").

CP(x) & CP(y) & x != y
  -> x ^ y = NP & x v y = Y # label("cps_noncomparable").
exists x (CP(x)).
```

These look similar to the axioms already given. The designation of `CounterProposal` is again a predicate.

Continuing with the set of assumptions, we define the basic combinators negation and equality:

```
% negation
~~x = x.
~A = N.
~N = A.
~Y = NP.
~NP = Y.
```

```
CP(x) <-> ~x = x.
%equality
x = y <-> x == y = A.
x != y <-> x == y = N.
x == y = y == x.
end_of_list.
```

And thus ends the base list of assumptions. Additional sets of assumptions can be included in separate files and used when the Prover9 or Mace4 programs are run from the command line.

Using these assumptions, we are able to prove or disprove formulas based on our SCC model. An example goal would be distributivity:

```
formulas(goals).
  (x v y) ^z = (x^z) v (y^z)     # label("distributivity").
end_of_list.
```

Using Mace4 we could verify that distributivity does not hold by letting it find a counterexample.

4.3.1.2 Experience and Results

If a counterexample exists, Mace4 will usually find it immediately. This was the case for distributivity (result shown in the appendix). On the other hand, proofs are more difficult to find and Prover9 will often search for longer than 10 or 15 minutes without a result. Because of this, more complex or general formulae must be broken down and proved granularly. The proof of De Morgan's Law is an example of this. An attempt to prove the general formula did not find a result within 30 minutes:

```
formulas(goals).
  ~(x ^ y) = ~x v ~ y      # label("de_morgan's").
```

```
end_of_list.
```

Before we could begin to prove De Morgan's Law a second time, we started off attempting to prove a relation:

$$x \leq y \iff \neg y \leq \neg x$$

There is an intuitive sense that it is true: negated constants are in the same position but on the opposite end of the lattice as the original constant, while counterproposals remain stationary in the middle. Negating both variables is equivalent to rotating the lattice 180 degrees, which corresponds to rotating the variables around the $\leq$ relation. Prover9, however, does not have this intuitive sense and fails to find a proof in adequate time.

We add the relation as an assumption in order to help prove more intermediary formulas—the general law is still too hard for Prover9 to prove directly. We use implication to break the search up into to separate searches (e.g. checking if the law holds when $x \leq y$ or $x \neq y$, etc.) and if the implications exhaustively cover all possible cases, then the law is true in general. The cases we choose are $x \leq y$, $y \leq x$, and $CP(x)$ & $CP(y)$, which is exhaustive because the only time two votes are not comparable using $\leq$ is when they are both counterproposals.

The two separate goals we prove are for when the variables are comparable and when they are non-comparable:

```
formulas(goals).
x <= y | y <= x -> ~ (y v x) = ~ y ^ ~ x # label("de morgan's comparable
    ")
end_of_list.
formulas(goals).
CP(x) & CP(y) -> ~ (y v x) = ~ y ^ ~ x # label("de morgan's non-
    comparable")
```

```
end_of_list.
```

The proofs for these can be found in the appendix.

4.3.2 TPTP and E Prover

A second automated theorem prover, E Prover, was used to compare performance with Prover9 and see if goals that could be proved in a reasonable time [15]. To make the comparison easier, a tool that was included with Prover9 was used to convert Prover9 inputs into the TPTP format [17], which is a commonly implemented format for automated theorem provers and is understood by E Prover. The converted inputs were then run on E Prover. Overall, it fared no better on proving any of the goals that Prover9 also had difficulty proving.

CHAPTER 5

ANALYSIS OF COMBINATOR SIZE

The amount of space needed to express or implement SDCs and SCCs based on their table representation will depend greatly on the arity of the combinator. As the number of leaf policies increases, the amount of space required will also increase, though the growth is not as easily described as that of functions from boolean or multi-valued logic. A boolean function with an arity of k will have 2^k rows in its table representation, with each row corresponding to a permutation of input values. Similarly, functions of multi-valued logic will have m^k rows, where m is the number of truth-values in the logic. The number of rows for an SDC will generally be less than a multi-valued function with the same number of truth-values as it has input-values, due to the restricted labelling of counter proposals within input permutations. The restriction causes some permutations to be invalid (e.g. (CP_2, CP_1) or (CP_2, CP_2)), resulting in those rows being omitted from the SDC table.

Due to the restricted labelling of counter proposals, the problem of counting the rows in an SCC table is closely related the problem of counting the ways a set can be partitioned. Counter proposals are labelled from $1, ..., n$ based on their occurrence in the input sequence and their distinctness from previous counter proposals. When counter proposals are given the same label for being equivalent, it is as if they are placed into the same subset of a partition of the input sequence. Looking at it this way, the label given to a counter proposal is the position of the first counter proposal added to the subset. A similar system for describing set partitions shows a correspondence with counter proposal labels.

When a set is partitioned into disjoint subsets called blocks, each element is placed in only one block. The mapping from elements to blocks can be represented using *Restricted Growth Strings* (RGSs) [13]. If we restrict the domain of SCCs to counter proposals, the permutations of input values are equivalent to RGSs. We describe the equivalence below.

RGSs always start with 1, since the first element will be placed in the first block. The next element will either be in the first block, in which case the string will have another 1, or in a new block, in which case the string will have a 2. Each successive element must be placed in a previous block or in a new one, so the RGS will not ever have a number that is larger than a previous number plus one. This is equivalent to our restriction on labelling counter proposals, that a counter proposal is only labelled CP_{i+1} if it is not equal to the previous $CP_1, ..., CP_i$.

Below are partitions of the set $\{a, b, c, d\}$, where a.bc.d is the partition $\{\{a\}, \{b, c\}, \{d\}\}$. Each subset is written in increasing order of its elements, and the subsets are listed in increasing order of their smallest element.

1 $blocks$: $abcd$

2 $blocks$: $abc.d$, $abd.c$, $acd.b$, $a.bcd$, $ab.cd$, $ac.bd$, $ad.bc$

3 $blocks$: $a.b.cd$, $a.bd.c$, $a.bc.d$, $ad.b.c$, $ac.b.d$, $ab.c.d$

4 $blocks$: $a.b.c.d$

The corresponding restricted growth strings for these partitions are:

1 $blocks$: 1111

2 $blocks$: 1112, 1121, 1211, 1222, 1122, 1212, 1221

3 $blocks$: 1233, 1232, 1223, 1231, 1213, 1123

4 $blocks$: 1234

The number in the n^{th} position of the RGS indicates which block the n^{th} element of the set (given some ordering, e.g. alphabetical in the above case) is placed in. So for the

partition $acd.b$, a, c, and d are in the first block and b is in the second block, resulting in an RGS of 1211.

An RGS can formally be defined as a string $a_1, ..., a_n$ with $a_1 = 1$ and

$$a_{i+1} \leq 1 + max(a_1, a_2, ..., a_i) \tag{5.1}$$

for $i = 1, 2, ..., n - 1$.

Bell numbers and *Stirling numbers* are well known sequences in combinatorics that can be used to count set partitions, and, accordingly, Restricted Growth Strings [13]. The same RGSs can be used to describe both the table for an SDC of arity n that only accepts counter proposals and the partitions of a set of n elements. The number of rows in the table and the number of partitions of the set will both equal the Bell number B_n. Table 5.1 shows the table representation of an SCC that corresponds to the previous set partition example.

Table 5.1. Input permutations for an SDC that only takes counter proposals. Numbers are used instead of the full CP_i notation to make the correspondence with RGSs more visible.

p_1	p_2	p_3	p_4
1	1	1	1
1	1	1	2
1	1	2	1
1	1	2	2
1	1	2	3
1	2	1	1
1	2	1	2
1	2	1	3
1	2	2	1
1	2	2	2
1	2	2	3
1	2	3	1
1	2	3	2
1	2	3	3
1	2	3	4

Stirling numbers of the second kind count the number of partitions with k blocks, and are given the notation

$$\left\{ n \atop k \right\} \tag{5.2}$$

In the set partition example, there were partitions with 1, 2, 3, and 4 blocks. Summing the number of partitions for each value of k will result in the Bell number for the set, or

$$\sum_{k=0}^{n} \left\{ n \atop k \right\} = B_n \tag{5.3}$$

For an SCC, which also accept constants Y, N, NP, and A as inputs, the Bell numbers no longer count the number of rows in its table, though a generalization of the Stirling numbers can be used to count them. The difference is that, unlike labelled counter proposals, constants are not restricted in what position they may appear within a permutation, breaking the correspondence with RGSs where all characters are restricted. However, given an RGS with a certain prefix, the characters in the prefix would appear unrestricted in the latter portion of the string. If a set of RGSs were truncated after a common prefix, they would all share a set of unrestricted characters that act as constants. *r-Stirling numbers of the second kind*, a generalization of Stirling numbers of the second kind, can be used to ensure the prefix of an RGS contains certain characters.

r-Stirling numbers of the second kind, as defined by Andrei [18], count the number of ways to partition a set of n elements using partitions of size k, such that the first r elements are in distinct subsets. They are given the notation:

$$\left\{ n \atop k \right\}_r \tag{5.4}$$

When the partitions counted by r-Stirling numbers of the second kind are represented as RGSs, the first r characters of the string will always be $1, 2, ..., r$, as each of the first r elements must be in distinct subsets. Since elements are partitioned in order, they will always be placed in the first r subsets. For example, if $r = 2$ the first two elements are placed in the first two subsets so that they are separate, resulting in a prefix of 12 for the RGS. The remaining $n - r$ elements are partitioned as they would with standard Stirling numbers of the second kind. Table 5.2 shows an example of restricted partitions.

Table 5.2. Restricted partitions of a set of four elements with corresponding RGSs, grouped by the number of subsets. The first two elements must be in distinct subsets.

2-Stirling number	Partition	RGS
$\left\{ {4 \atop 2} \right\}_2$	1.234	1222
	13.24	1212
	134.2	1211
	14.23	1221
$\left\{ {4 \atop 3} \right\}_2$	1.2.34	1233
	1.23.4	1223
	1.24.3	1232
	13.2.4	1213
	14.2.3	1231
$\left\{ {4 \atop 4} \right\}_2$	1.2.3.4	1234

In the portion of the string after the prefix, the numbers $1, ..., r$ will appear unrestricted, akin to constant values that appear unrestricted in the input permutations for SCCs. Restricting elements effectively makes them "constants" in the sense of SDC values when only considering the portion of the string that proceeds the prefix. The first r numbers will appear unrestricted in the truncated string while the characters $r + 1, ..., n$ will remain restricted, though with $r + 1$ behaving as the new first restricted number, allowing it to appear as the first character of the string.

Numbers $1, ..., r$ in a truncated RGS can be mapped to constants and $r+1, ..., n$ mapped to counter proposals $CP_1, ...CP_{n-r}$. Table 5.3 shows the mapping.

Table 5.3. The table shows the mapping between SDC input permutations and the highlighted portion of the RGSs.

$N \to 1$
$Y \to 2$
$CP_1 \to 3$
$CP_2 \to 4$

SCC input		RGS
p_1	p_2	
N	N	12**11**
N	Y	12**12**
N	CP_1	12**13**
Y	N	12**21**
Y	Y	12**22**
Y	CP_1	12**23**
CP_1	N	12**31**
CP_1	Y	12**32**
CP_1	CP_1	12**33**
CP_1	CP_2	12**34**

Theorem 5.1 *For SDCs of arity n with r constant values and unlimited counter proposals in their domain, the number of possible input permutations is equal to*

$$\sum_{k=r}^{n+r} \left\{ \begin{matrix} n+r \\ k \end{matrix} \right\}_r \tag{5.5}$$

for $n > 0$. When $n = 0$, the number of input permutations is r.

Proof As was detailed in this chapter, there is a direct mapping between the permutations of inputs for SDCs and the portion of RGSs after the prefix.

Permutations of SCC inputs may include constants (such as Y or N) that appear anywhere within the ordering and counter proposals ($CP_1, CP_2, ...$) that are restricted in where they can appear. A counter proposal labelled CP_j cannot appear in a permutation before CP_i if $j > i$.

RGSs that represent restricted partitions counted by r-Stirling numbers of the second kind will have a prefix that contains the characters $1, ..., r$. In the portion of the RGS that occurs after the prefix, these characters appear unrestricted, akin to constants that appear unrestricted in input permutations of SDCs. Characters that do not appear in the prefix,

$r+1, ..., r+n$, are subject to the standard restriction where $r+j$ cannot appear before $r+i$ in the string if $j > i$, which corresponds to the restriction on counter proposals.

So, for SCCs, which include Y, N, NP, and A as constants in their domain, r is set to four.

Theorem 5.2 *For SCCs with an arity of n, the number of possible input permutations, or rows in their table definition, is equal to*

$$\sum_{k=4}^{n+4} \left\{ \begin{matrix} n+4 \\ k \end{matrix} \right\}_4 \tag{5.6}$$

for $n > 0$. When $n = 0$, the number of input permutations is 4.

Proof Theorem 5.2 is a special case of Theorem 5.1 with r set to four.

CHAPTER 6

NUMBER OF POSSIBLE COMBINATORS

To get a sense of how large the space of combinators is and how it grows as the arity increases, we can define a formula for calculating just how many exist. To determine this, we can count how many distinct ways the input permutations, as enumerated in the previous chapter, can be mapped onto results. Each mapping from the set of input permutations to an output is a distinct combinator, or as they are referred to in logic, a truth function. With logic and with combinators these mappings can be visualized as the columns in the truth table, where each column represents a truth function. For example, boolean logic has 16 different binary operators, listed in the truth table below.

Table 6.1. The 16 boolean operators with an arity of 2.

p	q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

SDCs are a bit different because the domain and codomain grow with the arity, due to the existence of counterproposals.

6.1 Counting Combinators

Counting SCC-like combinators is a special case of counting permutations that allow repetition. For permutations with repetition, there are a number of items to choose from,

a, and a number of choices to make, s, from which the total number of ways to make s choices of a items is calculated by multiplying $a \times a \times a \times ...$ (s times) or a^s. For example, the boolean operators in Table 6.1 have two return values to chose from (0 or 1), so $a = 2$. A return value must be chosen for each permutation of inputs (00, 01, 10, and 11), so the number of choices to make is four, or $s = 4$. This means there is a total of 2^4, or 16 ways to make a boolean operator.

For combinators, with each assignment of input variables a "choice" must also be made of what value the combinator will return, so s will equal the number of permutations of input votes. The number of return values available to choose from, a, depends on if there are counterproposals in the input (since counterproposals cannot be returned unless they are present as an input value) and the number of constant votes in the environment, such as Y, N, or A.

Table 6.2. A 2-ary combinator with a domain and codomain of $\{Y, N, CP_1, CP_2\}$ has 10 possible variable assignments, each corresponding to a row ($s = 10$). Each row has a number of values it can return.

p_1	p_2	i (row number)	a_i (number of available return values)
N	N	1	2
N	CP_1	2	3
N	Y	3	2
CP_1	N	4	3
CP_1	CP_1	5	3
CP_1	CP_2	6	4
CP_1	Y	7	3
Y	N	8	2
Y	CP_1	9	3
Y	Y	10	2

Because a depends on the row, it will not be the same number multiplied each time as in the normal case of permutations with repetition where the total is always a^s. What works for boolean operators will not work SDCs. Rather, for SDCs there is some value a_i for each row $i \in 1, ..., s$ where a_i is the number of result values available for row i. The equation to

calculate the total number of permutations then becomes

$$a_1 \times a_2 \times ... \times a_s \tag{6.1}$$

There are still s numbers being multiplied, each representing a number of items available to choose from. From Table 6.2 we can determine equation 6.1 for 2-ary combinators with a domain and codomain of $\{Y, N, CP_1, CP_2\}$ to be $2 \times 3 \times 2 \times 3 \times 3 \times 4 \times 3 \times 2 \times 3 \times 2$, which can be rewritten as

$$(2^4)(3^5)(4^1)$$

It is important to note that the exponents not only add up to s (the number of rows, or "choices" to make) but that they correspond to r-Stirling numbers of the second kind. In Chapter 5 it was demonstrated that the number of rows of a combinator's truth table is equal to the sum of the r-Stirling numbers of the second kind over the appropriate range, so this is no coincidence. There is a connection between combinator input values and partitions counted by r-Stirling numbers of the second kind that was not highlighted in the previous chapter.

Recall that r-Stirling numbers of the second kind count the number of ways to partition a set of size n into k subsets, such that the first r elements are in different subsets (referred to as *restricted partitions*). Also recall that Restricted Growth Strings describe which subset each element of the set is partitioned into, so an RGS of 1211 means the first element is in the first partition, the second element is in the second partitions, and the third and fourth elements are in the first partition. RGSs make the mapping between restricted partitions and combinator input permutations more direct, as exemplified in Table 5.3, where each row of the truth table has a corresponding RGS. Intuitively, the subsets of a restricted partition correspond to the available values a combinator could return for the corresponding row, with the first r subsets being mapped to constant values (Y, N, NP, etc.), and the re-

maining subsets being mapped to counterproposals. Because they are restricted partitions, there are always r separate subsets available for selection, just as for combinators there are always r constants available as a return value.

To find the number of rows with a particular number of return values to choose from (k, e.g. we are finding the rows $i \in 1..s$ where $a_i = k$) it follows we only need to find the number of partitions with k subsets, which we can use the r-Stirling numbers of the second kind to find. In general, the number of rows with k possible return values is

$$\left\{ \begin{matrix} n+r \\ k \end{matrix} \right\}_r$$

Using the example from Table 6.2, we have 2 constant values Y and N ($r = 2$), and an arity of 2 ($n = 2$), so to find how many rows have 2, 3, and 4 possible return values we can use r-Stirling numbers of the second kind instead of counting by hand

$$\left(2^{\left\{ \begin{matrix} 4 \\ 2 \end{matrix} \right\}_2}\right)\left(3^{\left\{ \begin{matrix} 4 \\ 3 \end{matrix} \right\}_2}\right)\left(4^{\left\{ \begin{matrix} 4 \\ 4 \end{matrix} \right\}_2}\right) = (2^4)(3^5)(4^1)$$

When calculated, the r-Stirling numbers of the second kind give the same exponents that were counted by hand.

So, for each row there are k values to choose from and the total number of permutations for those rows is determined by multiplying k by itself $\left\{ \begin{matrix} n+r \\ k \end{matrix} \right\}_r$ times, or $k^{\left\{ \begin{matrix} n+r \\ k \end{matrix} \right\}_r}$. Since the number of available return values must be between r (only constants) and $r + n$ (constants and counter proposals) for each row, calculating $k^{\left\{ \begin{matrix} n+r \\ k \end{matrix} \right\}_r}$ for each value of k

from $r, ..., r + n$ and multiplying them together will give the total number of permutations across all rows.

Theorem 6.1 *The number of distinct SDCs of arity n with r constant values and unlimited counter proposals in their domain is counted by*

$$\prod_{k=r}^{n+r} k \left\{ \begin{matrix} n+r \\ k \end{matrix} \right\}_r \tag{6.2}$$

for $n \geq 0$, $r \geq 0$.

Proof SDCs map each permutation of input variable assignments to a result. Results are limited to constants and values that occur as input values. s, the total number of rows, choices must be made, and there are $a_1, ..., a_s$ items to choose from at each selection. The total number of ways to make these selections is

$$a_1 \times a_2 \times ... \times a_s$$

There are between r (just constants) and $r + n$ (constants and counter proposals) return values to choose from for each row, so

$$r \leq a_i \leq r + n$$

for $i \in \{1, ..., s\}$. To calculate how many rows have k return values, we use the r-Stirling number of the second kind, $\left\{ \begin{matrix} n+r \\ k \end{matrix} \right\}_r$. Since all rows have a number of return values between r and $r + n$, and we can calculate just how many rows do using r-Stirling number of the second kind, we can arrive at the final formula by multiplication.

Theorem 6.1 can be used to determine how many possible SCCs there are by setting the number of constants to four.

Theorem 6.2 *The number of distinct SCCs with an arity of n is counted by*

$$\prod_{k=4}^{n+4} k \left\{ \begin{matrix} n+4 \\ k \end{matrix} \right\}_4 \tag{6.3}$$

for $n \geq 0$.

Proof Theorem 6.2 is a special case of Theorem 6.1 where the number of constants, r, is set to 4.

6.2 Calculating the Number of SCCs

We can use Theorem 6.2 to calculate just how many different SCCs a policy writer could have available. To help with the calculations, some 4-restricted Stirling numbers of the second kind are listed in Table 6.3.

Table 6.3. 4-restricted Stirling numbers of the second kind [1] appear as exponents in instances of equation 6.3. The arity (n) of the combinator determines which row of integers will appear as exponents, e.g. if the arity is 2, the row indexed by 6 is used because $2 + 4 = 6$.

n+4 k	4	5	6	7	8	9
4	1					
5	4	1				
6	16	9	1			
7	64	61	15	1		
8	256	369	151	22	1	
9	1024	2101	1275	305	30	1

For an SCC with an arity of two ($n = 2$), Theorem 6.2 gives us the equation

$$\prod_{k=4}^{6} k^{\left\{ {6 \atop k} \right\}_4} = \left(4^{\left\{ {6 \atop 4} \right\}_4}\right)\left(5^{\left\{ {6 \atop 5} \right\}_4}\right)\left(6^{\left\{ {6 \atop 6} \right\}_4}\right)$$

Table 6.3 can be used to fill in the 4-restricted Stirling numbers of the second kind (using the row where $n + 2$ is 6 and k is 4, 5, and 6) that appear as exponents, producing a simplified equation:

$$(4^{16})(5^{9})(6^{1}) = 5.0331648 \times 10^{16}$$

Table 6.3 can be used again to produce the equation for SCCs with an arity of 3:

$$(4^{64})(5^{61})(6^{15})(7^{1}) = 4.85709396 \times 10^{93}$$

The number of possible combinators increases rapidly once the arity exceeds 1. The table below compares the number of SCCs with the number of boolean, 4, and 5-valued logic operators.

Table 6.4. The number of SCCs increases at a much greater rate with arity than boolean or 4-valued logic functions. The disparity arises because the number of counter proposals grows along with the arity, increasing the size of the domain and codomain for SCCs. The rate of growth as a function of arity is on the order of $O(n^{n^n})$. The number of SCCs with an arity of 4 has 550 digits in its decimal expansion, but can be written $((4^{256})(5^{369})(6^{151})(7^{22})(8^{1}))$.

Arity	SCCs	Boolean functs.	4-valued functs.	5-valued functs.
0	4	2	4	5
1	1280	4	256	3125
2	5.0331648×10^{16}	64	4294967296	$2.98023224 \times 10^{17}$
3	$4.85709396 \times 10^{93}$	256	4^{64}	2.3509887×10^{87}

CHAPTER 7

CONCLUSION

This paper proposes a general model for composing security policies for runtime enforcement mechanisms that are capable of emitting not only `Yes` and `No`, but arbitrary system events. This chapter summarizes the contributions and describes possible extensions to our model.

7.1 Summary

Static Decision Combinators provide a way to specify larger, complex policies from a number of smaller, base subpolicies. This allows for modular development and reuse of policy logic. Base policies are also abstract black-boxes so that combinators can be decoupled from the underlying implementation. We also provide a number of reusable general purpose combinators, such as *if-then-else*, *try-with*, and *dominates* as well as blueprints for constructing more unique ones, such as the *majority* and *quorum* combinators. The model also has the advantage of being amendable to analysis, which we have shown by detailing the algebraic properties of conjunction, disjunction, and negation combinators and showing how properties about these combinators can be proved using automated tools. Finally, we analyse the size of combinator truth tables and the breadth of combinators that exist.

7.2 Extensions

We present a couple of extensions to the ideas of policy composition presented in this thesis.

7.2.1 Tagged CounterProposals

`CounterProposals` are opaque to combinators, making it impossible to favor one over another based on the impact of the event. It may be desirable to give combinators basic knowledge on the nature of the event being proposed, which would allow policy authors to create more expressive combinators statically.

The policy author could define a set of tags that would be used to label `CounterProposals`. A policy would cast a `CounterProposal` vote with a tag and event tuple: $CP(tag, e)$. This would allow combinators to make different decisions based on the tags of `CounterProposals`. This increased expressiveness would also cause the size of the truth table definition to expand dramatically, depending on how many tags are added. Each tag/event combination would be a possible input, rather than just events as counterproposals.

7.2.2 Combination Algebra

Another possible interpretation of the negation combinator leads to a boolean algebra for composing policy decisions. In our interpretation, the negation of a counterproposal returns the same counterproposal. The algebraic model might define a *dual* vote to counterproposals, which we label $\overline{CP(e)}$. The meaning of the dual is "as long is the event is not e, I do not care". Naturally, the conjunction of $\overline{CP(e)}$ and $CP(e)$ is `No`, since they directly conflict. By their dual nature the disjunction would be `Abstain`.

The algebraic nature arises if we think about the values as sets, with a counterproposal being a set of just one event $\{e\}$, and the dual being the set of all possible votes excluding e. We can continue to write $\overline{CP(e)}$ as a shorthand to avoid writing an unbounded number of events (with the knowledge that the actual number of counterproposals during a query is limited by the number of subpolicies a policy has). In this interpretation, conjunction and disjunction become intersection and union on sets. It follows that `No` is actually the empty set and `Abstain` is the set of all possible votes. Each vote is essentially a set of all the outcomes the policy is okay with in response to the incoming proposal.

The vote of `Yes` in this model would essentially be the same as a counterproposal, except the event in its set is the incoming proposal. The dual of `Yes` in this case would not be the empty set `No` but rather the set of all events that are not the incoming proposal. This is similar to throwing an exception, which indicates the current proposal is invalid but does not preclude the insertion of counterproposals.

With every vote having a dual, conjunction, disjunction and negation combinators become operators in a boolean algebra, allowing for additional properties such as distributivity and De Morgan's Law to hold.

If the vote at the top of the policy does not include the incoming proposal, it would be interpreted as an exception and if it were the empty set, the system would halt because there were no valid events. However, this system would introduce additional issues to resolve, such as what to do when multiple counterposals reach the top. It may be reasonable so choose the first since all are deemed valid.

LIST OF REFERENCES

[1] Peter Bala. A143496 4-restricted stirling numbers of the second kind, August 2008. `http://oeis.org/A143496`.

[2] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 5(1):1–35, 2002.

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

[4] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[5] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, 2003.

[6] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, June 2011.

[7] Philip W. L. Fong and Ida Siahaan. Relationship-based access control policies and their policy languages. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 51–60, New York, NY, USA, 2011. ACM.

[8] G.A. Grätzer. *General lattice theory*. Birkhäuser, 2003.

[9] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):19, 2009.

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

[11] Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 87–100, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[13] Frank Rusky. Info about set partitions, March 2006. `http://theory.cs.uvic.ca/inf/setp/SetPartitions.html`.

[14] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[15] S. Schulz. E-a brainiac theorem prover. *AI Communications*, 15(2, 3):111–126, 2002.

[16] Stern. *Semimodular Lattices: Theory and Applications*. Dolciani Mathematical Expositions. The Mathematical Association of America, 1999.

[17] Geoff Sutcliffe and Christian Suttner. The tptp problem library for automated theorem proving. `http://www.cs.miami.edu/~tptp/`, 2001–2011.

[18] Andrei Z and Broder. The r-stirling numbers. *Discrete Mathematics*, 49(3):241 – 259, 1984.

APPENDICES

Appendix A SCC Definitions

The complete truth tables for the combinators introduced in previous chapters are displayed here.

Table A.1. Standard conjunction.

$\wedge$	N	NP	CP_1	CP_2	Y	A
N	N	N	N	N	N	N
NP	N	NP	NP	NP	NP	NP
CP_1	N	NP	CP_1	NP	CP_1	CP_1
Y	N	NP	CP_1	CP_2	Y	Y
A	N	NP	CP_1	CP_2	Y	A

Table A.2. Standard disjunction.

$\vee$	N	NP	CP_1	CP_2	Y	A
N	N	NP	CP_1	CP_2	Y	A
NP	NP	NP	CP_1	CP_2	Y	A
CP_1	CP_1	CP_1	CP_1	Y	Y	A
Y	Y	Y	Y	Y	Y	A
A	A	A	A	A	A	A

Table A.3. Equality.

$=$	N	NP	CP_1	CP_2	Y	A
N	A	N	N	N	N	N
NP	N	A	N	N	N	N
CP_1	N	N	A	N	N	N
Y	N	N	N	N	A	N
A	N	N	N	N	N	A

Appendix A (Continued)

Table A.4. Inequality.

$\neq$	N	NP	CP_1	CP_2	Y	A
N	N	A	A	A	A	A
NP	A	N	A	A	A	A
CP_1	A	A	N	A	A	A
Y	A	A	A	A	N	A
A	A	A	A	A	A	N

Appendix A (Continued)

Table A.5. if-then-else

p_2 p_3 \ p_1		N	NP	CP_1	Y	A
N	N	N	N	N	N	N
N	NP	NP	NP	NP	N	N
N	CP_1	CP_1	CP_1	CP_1	N	N
N	CP_2	CP_2	CP_2	CP_2	N	N
N	Y	Y	Y	Y	N	N
N	A	A	A	A	N	N
NP	N	N	N	N	NP	NP
NP	NP	NP	NP	NP	NP	NP
NP	CP_1	CP_1	CP_1	CP_1	NP	NP
NP	CP_2	CP_2	CP_2	CP_2	NP	NP
NP	Y	Y	Y	Y	NP	NP
NP	A	A	A	A	NP	NP
CP_1	N	N	N	N	CP_1	CP_1
CP_1	NP	NP	NP	NP	CP_1	CP_1
CP_1	CP_1	CP_1	CP_1	CP_1	CP_1	CP_1
CP_1	CP_2	CP_2	CP_2	CP_2	CP_1	CP_1
CP_1	Y	Y	Y	Y	CP_1	CP_1
CP_1	A	A	A	A	CP_1	CP_1
CP_2	N	N	N	N	CP_2	CP_2
CP_2	NP	NP	NP	NP	CP_2	CP_2
CP_2	CP_1	CP_1	CP_1	CP_1	CP_2	CP_2
CP_2	CP_2	CP_2	CP_2	CP_2	CP_2	CP_2
CP_2	CP_3	CP_3	CP_3	CP_3	CP_2	CP_2
CP_2	Y	Y	Y	Y	CP_2	CP_2
CP_2	A	A	A	A	CP_2	CP_2
Y	N	N	N	N	Y	Y
Y	NP	NP	NP	NP	Y	Y
Y	CP_1	CP_1	CP_1	CP_1	Y	Y
Y	CP_2	CP_2	CP_2	CP_2	Y	Y
Y	Y	Y	Y	Y	Y	Y
Y	A	A	A	A	Y	Y
A	N	N	N	N	A	A
A	NP	NP	NP	NP	A	A
A	CP_1	CP_1	CP_1	CP_1	A	A
A	CP_2	CP_2	CP_2	CP_2	A	A
A	Y	Y	Y	Y	A	A
A	A	A	A	A	A	A

Appendix B Prover9 and Mace4 Inputs and Proofs

The main input file `scc.in`:

```
% Language Options
op(325, prefix, "~").
op(600, infix, "==").

if(Prover9). % Options for Prover9
  assign(max_seconds, 120).
end_if.
if(Mace4).   % Options for Mace4
  assign(max_seconds, 60).
end_if.

formulas(assumptions).
% Lattice Theory
x ^ y = y ^ x                 # label("commutativity_meet").
x v y = y v x                 # label("commutativity_join").
(x ^ y) ^ z = x ^ (y ^ z)     # label("associativity_meet").
(x v y) v z = x v (y v z)     # label("associativity_join").
(x v y) ^ x = x               # label("absorption_1").
(x ^ y) v x = x               # label("absorption_2").

% define relations
x <= y <-> x ^ y = x          # label("less_than_equal").
x < y <-> x <= y & x != y     # label("less_than").

% SCC votes
x v N = x                     # label("join_identity").
x ^ A = x                     # label("meet_identity").
```

Appendix B (Continued)

```
N < NP.
Y < A.
N < x -> NP <= x              # label("atomic_element").
x < A -> x <= Y               # label("coatomic_element").

% counterproposals
CP(x) <-> -(x = Y
          | x = NP |
            x = A |
            x = N)            # label("counter_proposal").

CP(x) & CP(y) & x != y
   -> x ^ y = NP & x v y = Y # label("cps_noncomparable").

exists x (CP(x)).

% negation
~~x = x.
~A = N.
~N = A.
~Y = NP.
~NP = Y.
CP(x) <-> ~x = x.

x = y <-> x == y = A.
x != y <-> x == y = N.
x == y = y == x.
end_of_list.
```

Appendix B (Continued)

B.1 Proofs

Proofs include the goal, additional assumptions, and the proof or counterexample that was found.

B.1.1 Distributivity Counterexample

A disproof of distributivity was found using Mace4. The goal as it was input:

```
formulas(goals).
  (x v y) ^z = (x^z) v (y^z)     # label("distributivity").
end_of_list.
```

The model found as a counterexample:

```
% number = 1
% seconds = 0

% Interpretation of size 7

 == :
        | 0 1 2 3 4 5 6
    ----+---------------
      0 | 0 1 1 1 1 1 1
      1 | 1 0 1 1 1 1 1
      2 | 1 1 0 1 1 1 1
      3 | 1 1 1 0 1 1 1
      4 | 1 1 1 1 0 1 1
      5 | 1 1 1 1 1 0 1
      6 | 1 1 1 1 1 1 0
```

Appendix B (Continued)

< :

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	1	0	1	1	1	1	1
2	1	0	0	1	1	1	1
3	1	0	0	0	0	0	0
4	1	0	0	1	0	0	0
5	1	0	0	1	0	0	0
6	1	0	0	1	0	0	0

<= :

	0	1	2	3	4	5	6
0	1	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	1	0	1	1	1	1	1
3	1	0	0	1	0	0	0
4	1	0	0	1	1	0	0
5	1	0	0	1	0	1	0
6	1	0	0	1	0	0	1

^ :

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	1	1	1	1	1	1
2	2	1	2	2	2	2	2
3	3	1	2	3	4	5	6

Appendix B (Continued)

```
     4  |  4  1  2  4  4  2  2
     5  |  5  1  2  5  2  5  2
     6  |  6  1  2  6  2  2  6

v :
        |  0  1  2  3  4  5  6
    ----+-------------------
     0  |  0  0  0  0  0  0  0
     1  |  0  1  2  3  4  5  6
     2  |  0  2  2  3  4  5  6
     3  |  0  3  3  3  3  3  3
     4  |  0  4  4  3  4  3  3
     5  |  0  5  5  3  3  5  3
     6  |  0  6  6  3  3  3  6

A : 0

N : 1

NP : 2

Y : 3

c1 : 4

c2 : 4

c3 : 5
```

Appendix B (Continued)

```
c4 : 6

~ :
        0 1 2 3 4 5 6
    ------------------
        1 0 3 2 4 5 6

CP :
        0 1 2 3 4 5 6
    ------------------
        0 0 0 0 1 1 1
```

The result shows that if there are three different counterproposals, the meet and join SCCs will not distribute.

B.1.2 De Morgan's Law

Two goals are used to prove De Morgan's Law. The proofs for each goal are listed below. Each proof used an additional assumption:

```
formulas(assumptions).
  x <= y <-> ~y <= ~x.
end_of_list.
```

The first goal is for non-comparable votes (`CounterProposals`):

```
formulas(goals).
  CP(x) & CP(y) -> (~(x v y) = ~x ^ ~ y)       # label("de_morgan's").
end_of_list.
```

The proof:

Appendix B (Continued)

```
% ———— Comments from original proof ————
% Proof 1 at 5.16 (+ 0.32) seconds.
% Length of proof is 37.
% Level of proof is 9.
% Maximum clause weight is 12.
% Given clauses 1595.

1 x <= y <-> x ^ y = x # label(non_clause).  [assumption].
5 CP(x) <-> -(x = Y | x = NP | x = A | x = N) # label("counter_proposal"
    ) # label(non_clause).  [assumption].
6 CP(x) & CP(y) & x != y -> x ^ y = NP & x v y = Y # label(non_clause).
     [assumption].
8 CP(x) <-> ~ x = x # label(non_clause).  [assumption].
13 CP(x) & CP(y) -> (x <= y <-> ~ y <= ~ x) # label(non_clause) # label(
    goal).  [goal].
14 x ^ y = y ^ x # label("commutativity_meet").  [assumption].
15 x v y = y v x # label("commutativity_join").  [assumption].
18 (x v y) ^ x = x # label("absorption_1").  [assumption].
19 x ^ (x v y) = x.  [copy(18),rewrite([14(2)])].
20 (x ^ y) v x = x # label("absorption_2").  [assumption].
21 x v (x ^ y) = x.  [copy(20),rewrite([15(2)])].
22 -(x <= y) | x ^ y = x.  [clausify(1)].
23 x <= y | x ^ y != x.  [clausify(1)].
34 -CP(x) | NP != x # label("counter_proposal").  [clausify(5)].
38 -CP(x) | -CP(y) | y = x | x ^ y = NP.  [clausify(6)].
46 -CP(x) | ~ x = x.  [clausify(8)].
54 CP(c2).  [deny(13)].
55 CP(c3).  [deny(13)].
```

Appendix B (Continued)

```
56 c2 <= c3 | ~ c3 <= ~ c2.  [deny(13)].
57 -(c2 <= c3) | -(~ c3 <= ~ c2).  [deny(13)].
70 x ^ x = x.  [para(21(a,1),19(a,1,2))].
129 ~ c2 = c2.  [resolve(54,a,46,a)].
132 -CP(x) | c2 = x | x ^ c2 = NP.  [resolve(54,a,38,b)].
136 c2 != NP.  [resolve(54,a,34,a),flip(a)].
138 -(c2 <= c3) | -(~ c3 <= c2).  [back_rewrite(57),rewrite([129(7)])].
139 c2 <= c3 | ~ c3 <= c2.  [back_rewrite(56),rewrite([129(7)])].
140 ~ c3 = c3.  [resolve(55,a,46,a)].
147 c3 != NP.  [resolve(55,a,34,a),flip(a)].
149 c2 <= c3 | c3 <= c2.  [back_rewrite(139),rewrite([140(5)])].
150 -(c2 <= c3) | -(c3 <= c2).  [back_rewrite(138),rewrite([140(5)])].
162 x <= x.  [resolve(70,a,23,b)].
249 c2 <= c3 | c2 ^ c3 = c3.  [resolve(149,b,22,a),rewrite([14(6)])].
468 c2 ^ c3 = c3 | c2 ^ c3 = c2.  [resolve(249,a,22,a)].
3354 c3 = c2 | c2 ^ c3 = NP.  [resolve(132,a,55,a),rewrite([14(6)]),flip
    (a)].
24692 c2 ^ c3 = c2 | c3 = c2.  [para(468(a,1),3354(b,1)),unit_del(c,147)
    ].
24786 c3 = c2.  [para(24692(a,1),3354(b,1)),merge(b),unit_del(b,136)].
24787 $F.  [back_rewrite(150),rewrite([24786(2),24786(4)]),merge(b),
    unit_del(a,162)].
```

The second goal, for comparable votes:

```
formulas(goals).
  x <= y | y <= x -> (~(x v y) = ~x ^ ~ y)      # label("de_morgan's").
end_of_list.
```

Appendix B (Continued)

The proof:

```
% ---------- Comments from original proof ----------
% Proof 1 at 34.39 (+ 2.39) seconds.
% Length of proof is 58.
% Level of proof is 9.
% Maximum clause weight is 12.
% Given clauses 6403.

1 x <= y <-> x ^ y = x # label("less_than_equal") # label(non_clause).
    [assumption].
2 x < y <-> x <= y & x != y # label("less_than") # label(non_clause). [
    assumption].
5 CP(x) <-> -(x = Y | x = NP | x = A | x = N) # label("counter_proposal"
    ) # label(non_clause). [assumption].
8 CP(x) <-> ~ x = x # label(non_clause). [assumption].
9 x = y <-> x == y = A # label(non_clause). [assumption].
10 x != y <-> x == y = N # label(non_clause). [assumption].
11 x <= y <-> ~ y <= ~ x # label(non_clause). [assumption].
12 x <= y | y <= x -> ~ (x v y) = ~ x ^ ~ y # label("de_morgan's") #
    label(non_clause) # label(goal). [goal].
13 x ^ y = y ^ x # label("commutativity_meet"). [assumption].
14 x v y = y v x # label("commutativity_join"). [assumption].
17 (x v y) ^ x = x # label("absorption_1"). [assumption].
18 x ^ (x v y) = x. [copy(17),rewrite([13(2)])].
19 (x ^ y) v x = x # label("absorption_2"). [assumption].
20 x v (x ^ y) = x. [copy(19),rewrite([14(2)])].
21 -(x <= y) | x ^ y = x # label("less_than_equal"). [clausify(1)].
22 x <= y | x ^ y != x # label("less_than_equal"). [clausify(1)].
```

Appendix B (Continued)

```
24 -(x < y) | y != x # label("less_than").  [clausify(2)].
25 x < y | -(x <= y) | y = x # label("less_than").  [clausify(2)].
27 x ^ A = x # label("meet_identity").  [assumption].
34 -CP(x) | A != x # label("counter_proposal").  [clausify(5)].
40 ~ ~ x = x.  [assumption].
41 ~ A = N.  [assumption].
42 ~ N = A.  [assumption].
46 CP(x) | ~ x != x.  [clausify(8)].
47 x != y | y == x = A.  [clausify(9)].
49 x = y | y == x = N.  [clausify(10)].
51 x == y = y == x.  [assumption].
52 -(x <= y) | ~ y <= ~ x.  [clausify(11)].
53 x <= y | -(~ y <= ~ x).  [clausify(11)].
54 c2 <= c3 | c3 <= c2 # label("de_morgan's").  [deny(12)].
55 ~ (c2 v c3) != ~ c2 ^ ~ c3 # label("de_morgan's").  [deny(12)].
60 x ^ (y v x) = x.  [para(14(a,1),18(a,1,2))].
64 x v (y ^ x) = x.  [para(13(a,1),20(a,1,2))].
68 x ^ x = x.  [para(20(a,1),18(a,1,2))].
77 x <= A.  [resolve(27,a,22,b)].
86 -CP(A).  [ur(34,b,27,a(flip)),rewrite([68(3)])].
105 N != A.  [para(41(a,1),46(b,1)),unit_del(a,86)].
111 x == x = A.  [resolve(47,a,40,a),rewrite([40(2)])].
116 -(x == y < y == x).  [ur(24,b,51,a)].
124 N <= x.  [para(42(a,1),53(b,2)),unit_del(b,77)].
131 c2 <= c3 | c2 ^ c3 = c3.  [resolve(54,b,21,a),rewrite([13(6)])].
139 N < x | N = x.  [resolve(124,a,25,b),flip(b)].
157 x <= y v x.  [resolve(60,a,22,b)].
184 ~ (x v y) <= ~ y.  [resolve(157,a,52,a)].
187 x ^ y <= x.  [para(20(a,1),157(a,2))].
```

Appendix B (Continued)

```
203 x ^ y <= y.  [para(13(a,1),187(a,1))].
206 ~ x <= ~ (y ^ x).  [resolve(203,a,52,a)].
226 x = y | -(N < x == y).  [para(49(b,1),116(a,1))].
611 ~ x ^ ~ (y v x) = ~ (y v x).  [resolve(184,a,21,a),rewrite([13(4)])
    ].
753 ~ x ^ ~ (y ^ x) = ~ x.  [resolve(206,a,21,a)].
956 c2 ^ c3 = c3 | c2 ^ c3 = c2.  [resolve(131,a,21,a)].
1278 -(N < ~ c2 ^ ~ c3 == ~ (c2 v c3)).  [ur(226,a,55,a),rewrite([51(11)
    ])].
1761 ~ c2 ^ ~ c3 == ~ (c2 v c3) = N.  [resolve(1278,a,139,a),flip(a)].
24532 c2 ^ c3 = c2 | c2 v c3 = c2.  [para(956(a,1),20(a,1,2))].
31954 c2 ^ c3 = c2.  [para(24532(b,1),611(a,1,2,1)),rewrite([13(10)]),
    flip(b),unit_del(b,55)].
31964 c2 v c3 = c3.  [para(31954(a,1),64(a,1,2)),rewrite([14(3)])].
32032 ~ c2 ^ ~ c3 = ~ c3.  [para(31954(a,1),753(a,1,2,1)),rewrite([13(5)
    ])].
32131 $F.  [back_rewrite(1761),rewrite([32032(5),31964(5),111(5)]),flip(
    a),unit_del(a,105)].
```