A Verified Algorithm for Detecting Conflicts in XACML Access Control Rules

Michel St-Martin

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of Master of Science in Mathematics [1]

Department of Mathematics and Statistics
Faculty of Science
University of Ottawa

© Michel St-Martin, Ottawa, Canada, 2012

[1] The M.Sc. program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

Abstract

The goal of this thesis is to find provably correct methods for detecting conflicts between XACML rules. A conflict occurs when one rule permits a request and another denies that same request. As XACML deals with access control, we can help prevent unwanted access by verifying that it contains rules that do not have unintended conflicts. In order to help with this, we propose an algorithm to find these conflicts then use the Coq Proof Assistant to prove correctness of this algorithm. The algorithm takes a rule set specified in XACML and returns a list of pairs of indices denoting which rules conflict. It is then up to the policy writer to see if the conflicts are intended, or if they need modifying. Since we will prove that this algorithm is sound and complete, we can be assured that the list we obtain is complete and only contains true conflicts.

Résumé

L'objectif de cette thèse est de trouver des méthodes qui détectent des conflits entre les règles dans XACML. On doit pouvoir démontrer que ces règles sont correctes. Un conflit se produit lorsqu'une demande d'accès est acceptée par une règle et refusée par une autre. Puisque XACML est utilisé pour définir des règles d'accès, on peut aider à éviter les accès non autorisés en s'assurant que les règles n'ont pas de conflits non voulus. Afin d'éviter ce problème, on propose un algorithme qui identifie ces conflits et par la suite on utilise "Coq Proof Assistant" afin de montrer que l'algorithme fonctionne comme voulu. L'algorithme prend un ensemble de règles et donne une liste de paires d'indices. Pour chaque paire dans la liste, il y a un conflit (les nombres dans la paire contiennent l'emplacement des règles en conflit). C'est ensuite à l'administrateur de vérifier si les conflits sont voulus ou s'ils doivent être modifiés. Puisqu'on va montrer que l'algorithme fonctionne bien dans tous les cas, on peut être assuré que cette liste est complète et que chaque paire est un vrai conflit.

Acknowledgements

I would like to offer my sincerest gratitude to my supervisor, Dr. Amy Felty, for her guidance, her advice, her encouragement as well as her numerous suggestions helping me improve the thesis.

I am also thankful for the funding received from the University of Ottawa during which time I was completing my thesis. With their financial aid, I was able to concentrate more on my work.

Thanks also go out to all those that have revised my thesis. Their time, effort and comments were greatly appreciated.

Finally, I want to thank everybody, especially my family, who encouraged me in pursuing my graduate studies. This experience has given me great satisfaction and worth.

Contents

List of Figures **vi**

List of Tables **viii**

1 Introduction **1**

1.1 Outline of thesis 2

2 Formal Proof **4**

3 Background **7**

3.1 XACML 7

3.2 Coq 12

4 Implementation **31**

4.1 Representing XACML in Coq 32

4.2 Intersection of *srac* 40

4.3 Conflict Detection 45

5 Algorithm Correctness **49**

5.1 Custom tactics 49

5.2 Soundness 56

5.3 Completeness 70

6 Expandability **78**

6.1 Adding new types 78

6.2 Adding a new *srac* 80

7 Related Work **84**

8 Conclusion and Future Work **86**

A Tactics.v **88**

B TimeRange.v **98**

C ConflictDetect.v **105**

D RuleSet.v **126**

E Example Policy **138**

F Testing the Algorithm **142**

G Testing in Coq **145**

Bibliography **147**

List of Figures

4.1 Definition of *valueMatch* 37
4.2 Time range chart 41
4.3 Cases for *timeInRange* 43
4.4 Definition of *sracCheck* 46

List of Tables

4.1 Intersection ranges of two *timeInRange* 43

Chapter 1

Introduction

In this modern age, security is a must. In this thesis, we will look at some complications in an access control tool called eXtensible Access Control Markup Language (XACML) standardized by OASIS [12]. XACML is a policy specification language that allows administrators to define access rules (which either permit or deny access to a certain group of people). When someone requires access, he makes a request. That request is then compared with each access rule to see which applies. Should more than one rule apply, each is considered along with all other rules that apply using the rule combining algorithm located in the header of the file. These combining algorithms include "permit overrides" and "first applicable". These algorithms often lead to consequences not intended by the policy writer. Should the writer forget about a rule higher up it the list, it could apply before the lower rules. This could give access to the wrong people, or it could deny access to someone who needs it.

We propose an algorithm for finding conflicts between these rules and report the conflicts to the policy writer. After using the algorithm, the policy writer can check each conflict to see if the conflicts were intended, or if rules need to be changed. The algorithm we propose will be formally verified to make sure the list of conflicts is complete and only contains true conflicts (soundness).

1.1 Outline of thesis

We use formal proofs, proofs directly following from a set of axioms, both with and without computer help, in mathematics in many ways. In Chapter 2, we will see examples of formal mathematics used in the past.

In Chapter 3, we offer some background material to introduce XACML [12] and the Coq Proof Assistant [4, 15]. We will present the general form of XACML rules and rule sets in Section 3.1. In Section 3.2, we will introduce Coq. In Coq, we will learn how to define objects, how to state lemmas and theorems and how to prove them.

In Chapter 4, we will construct the conflict algorithm. In Section 4.1, we will define rules, requests and conflicts using Coq's syntax. The conflict detection algorithm, which is a generalization of the algorithm found in [3], uses the notions of intersections to determine the set of requests that two rules apply to. Section 4.2 presents the Coq development for expressing intersections between rules for a subset of the full XACML language. Section 4.3 will use these intersections to define the conflict detection algorithm.

In Chapter 5, we will prove the correctness of the algorithm. In Section 5.1, we define custom tactics (commands which are used to simplify proofs) that are used in the proofs. In Section 5.2 we prove that the algorithm is sound, that is, each conflict it finds is truly a conflict. In Section 5.3 we prove the completeness of the algorithm, that is, that it finds every conflict.

Soundness is important because if the algorithm returns too many false positives, it wastes the policy writer's time. He probably has much better things to do than to sift through countless conflicts that don't even exist. Completeness is also important because if the algorithm doesn't report every conflict, a missing conflict could be one that needs changing. Proving correctness of algorithm often leads to finding errors in the algorithm. In the earlier stages of the algorithm, a case was missing for finding

conflicts between two time ranges. Because of this, it was impossible to prove the theorem for completeness. The missing case was then added, and the theorem was proved.

In Chapter 6, we propose a way to expand the definitions in order to allow the algorithm to work with more types and more functions, illustrating how the work can be extended to handle more of XACML in a modular way.

In Chapter 7, we look at some related work. We then conclude and offer some future work that can be done with respect to the algorithm in Chapter 8.

In Appendices A-D, we show the Coq code for the algorithm. Each appendix contains the source for one file.

We have written a proof of concept algorithm in Java which closely corresponds to the Coq algorithm. This uses the same methods as the Coq algorithm. Sun Microsystems had implemented an open-source version of XACML which included a parser [11]. Using this tool seemed to be the easiest way to test our algorithm on an actual policy file. An example policy was created specifically for testing intersections between time ranges. This example policy can be found in Appendix E. In Appendix F, we find the result of running the Java algorithm. In Appendix G, the Java version's parser was used to generate Coq code which was used to find conflicts on the same example.

My contributions for this thesis are:

- Coq definitions for XACML concepts found in Section 4.1.
- A conflict detection algorithm found in Sections 4.2 and 4.3.
- Tactics that can be used to simplify certain proofs found in Section 5.1.
- Proofs for correctness of the algorithm found in Sections 5.2 and 5.3.
- A way to expand the algorithm with new types and functions found in Chapter 6.

Chapter 2

Formal Proof

A common dictionary definition of proof is "Evidence sufficient to establish a thing as true, or to produce belief in its truth". But how much is "sufficient". At one extreme, we have a "hand waving" proof which essentially assumes the statement is true, and at the other extreme, we have **formal proof**. A formal proof is a series of sentences that is either an axiom, or that follows directly from previous sentences (using inference rules) eventually arriving at the statement we want to prove. We must arrive at this statement without making any assumptions (except our axioms) or miss any cases. If we establish a formal proof for a statement, we can be assured that the statement is indeed true.

At the beginning of the 20^{th} century, formal proof was a prime interest. A noteworthy example is that of Whitehead and Russell, who tried to formally prove many topics in mathematics, covering set theory, cardinal numbers, ordinal numbers, and real numbers in their three volume work titled Principia Mathematica [17, 18, 19]. These books laid a foundation for modern mathematics. We can also look at the works of Ernst Zermelo and Abraham Fraenkel who proposed the Zermelo-Fraenkel set theory with the axiom of choice (ZFC) [20], an axiomatic set theory in which it is believed that most if not all of mathematics could follow from its axioms augmented

with definitions.

In many applications of formal proofs, there are a large number of cases to check in order to prove a statement, often too high of a number to check by hand. In these cases, we resort to computers to help list and check each case. We take the four colour theorem by Appel and Haken [1] as an example. This was the first theorem that was proven heavily using computers. The theorem states that any planar surface separated into regions (forming a map) can be coloured with as few as four colours and have the property that any two regions in the map which share a border are never coloured the same colour. To prove this, they listed a set of 1,936 maps and showed that every map could be reduced to one of these 1,936 maps. This meant that any "smallest" counter example had to be in this set. They then showed that each of these maps could be coloured appropriately, thus showing there could be no counter example. Dealing with 1,936 maps would be infeasible to do by hand, and so they resorted to a computer, built specifically for this task, to aid in the proof.

This theorem was not generally accepted as there was no way to verify their work. There were also doubts of having computer bugs that would make the computer accept the theorem when in reality it is false. Many more attempts at proving this theorem have been made since then. One noteworthy attempt is by Georges Gonthier [9]. His approach was similar to that of Appel and Haken, however he used Coq [15], a general theorem prover to do so. His proof was then (machine) checked by the Coq compiler. By having Coq check his work, he minimized the chance that his theorem was wrong. This chance is minimal since Coq, as a general theorem prover, although not verified itself, implements inference rules of a formal logic and has been widely used and tested (as opposed to Appel and Haken's computer which was only used once). There is no general use computer, which has been proven to never be able to make mistakes. This sort of computer is impossible to make, as it would have to be so complex that it would be impossible to verify its "mathematical perfectness" without the use of other computers, which for the same reason has also not been

verified. Thus so we must be content to using computers and software which is more likely to crash completely than to accept a false theorem.

Another area of formal proof is that of proof-carrying code. Proof-carrying code, which was first described by George Necula and Peter Lee [13], is a mechanism which is used to verify certain properties about an algorithm. Examples of desirable properties are that the algorithm is safe (works without damaging a computer or leaking personal data), that it is correct (it works as intended in all cases), or that the algorithm ends (the algorithm never goes into an infinite loop, or leads to a never ending growth of cases). An algorithm that hasn't been proven correct is analogous to that hand waving proof mentioned earlier. Even if there is a chance that proof checking software/hardware is faulty, a proof-carrying algorithm is far more likely to be correct than another without proof, and is thus more mathematically sound.

In pure mathematics, some mathematicians trying to prove a theorem with many cases have resorted to doing a rough draft of a proof in conventional means then using a computer to formally verify the proof. For example, mathematician Vladimir Voevodsky is working on a univalent foundations project which is a new foundation for mathematics [16]. Voevodsky is actively developing Coq libraries to support his results. Verifying a proof gives more confidence in the proof as the computer will not miss any cases that might have been overlooked trying to prove something by hand, and never tires of checking a large number of cases. Having a machine check your proof can assure you of its correctness, but still allows that more intuitive proof. In the case of the four colour theorem above, much of the original lack of acceptance was due to the fact that the proof of the 1936 maps was done by hand. The proof was over 100 pages of work and hard to verify. The theorem was more widely accepted after these cases were formally checked by a computer.

With the intent of having a mathematically correct algorithm, this thesis will use formal proof (checked by Coq) to show the correctness of the algorithm developed here.

Chapter 3

Background

In this chapter, we present the background material needed to understand this thesis. This includes the XACML policy language (3.1) and the Coq Proof Assistant (3.2).

3.1 XACML

Short for eXtensible Access Control Markup Language, XACML is an access control policy language that uses XML to define its policies and requests. It is standardized by OASIS, and so can be used for many different types of access control requirements. In XACML, requests to use a resource are tested against policies to see if the given request is permitted or denied.

Policy files A XACML policy file is comprised of header information, a target, a list of one or more rules and a set of obligations. Since this thesis only deals with finding conflicts between rules, it will ignore the headers and the set of obligations, such as writing to access logs.

Subjects, Resources and Actions Subjects, resources and actions are lists of subjects that want access, resources requested, and actions the subject wants to

accomplish on the resource, respectively. The following is the general syntax for a resource. Note that anything between ... is not specifically part of the syntax, it simply describes what must be added at that location.

```
<Resources>
   <Resource>
      <ResourceMatch MatchId="...name of function...">
         ...Attributes needed...
      </ResourceMatch>
   </Resource>
   ...
   <Resource>
      <ResourceMatch MatchId="...name of function...">
        ...Attributes needed...
      </ResourceMatch>
   </Resource>
</Resources>
```

An example of the above is that of a user wishing to print, shown below. To save space and for readability, information which must be included in the full specification, in this case, match ids and the data types, have been shortened. E.g. “string-equals” is short for “urn:oasis:names:tc:xacml:1.0:function:string-equal”.

```
<Resources>
   <Resource>
      <ResourceMatch MatchId="string-equal">
         <AttributeValue DataType="string">printer</AttributeValue>
         <ResourceAttributeDesignator DataType="string"
                                      AttributeId="resource-id"/>
      </ResourceMatch>
   </Resource>
</Resources>
```

This says that to test the request against this resource, we should use the function “string-equals”, using the value “printer”, which is of type string, as an argument, then compare with the resource-id from the request.

The syntax for subjects and actions are nearly identical, obtained by replacing "Resource" by "Subject" or "Action" (this includes replacing words containing resource, such as ResourceMatch, by the corresponding term).

Should we need any subject to apply, we use the syntax:

```
<Subjects>
   <AnySubject/>
</Subjects>
```

If the entire <Subjects> block is omitted, the same result ensues. Similarly, we can get "any resource" or "any action".

For a requests subject to apply, it need only match one of the subjects in the policy. It is similar for resources and actions.

Targets The target is used in XACML to group subjects, resources and actions. Targets are found in two places. The first is in a policy. This target is global to the policy, and applies to everything in it. The second place is in a rule (which is in a policy). This target only applies to the one rule it occurs in. For a rule to apply to a given policy, both the target in the rule and the global target must apply. One can write <Target/> which means that this target applies to any request. Again, omitting the Target block entirely has the same effect. The syntax for a target is shown below:

```
<Target>
   <Subjects>
      ...Syntax for subject above...
   </Subjects>
   <Resources>
      ...Syntax above...
   </Resources>
   <Actions>
      ...Syntax above...
   </Actions>
</Target>
```

Rules A rule groups what to do (either permit or deny access) if the target applies, a target, and an optional condition. Conditions are similar to subjects, resources and actions, but allow functions that don't relate directly to those 3 fields, such as the time of day. Conditions need to apply for the rule to apply. They have similar syntax to subjects and are found outside of the target. An example of a condition would be: the rule only applies during business hours. Below is the syntax for a rule that permits requests:

```
<Rule RuleId="...Rule name..." Effect="Permit">
   <Target>
      ...Syntax above...
   <Target/>

   <Condition FunctionId="...name of function...">
      ...Attributes for condition...
   </Condition>
</Rule>
```

For rules that deny requests, "Permit" is replaced by "Deny".

We can make a "permit all" or a "deny all" rule using the syntax:

```
<Rule RuleId="...Rule name..." Effect="Permit"/>
<Rule RuleId="...Rule name..." Effect="Deny"/>
```

For a rule to apply, the target in the rule, the global target and all conditions (if any) must apply. If a rule applies, then it is considered along with all other rules that apply using the rule combining algorithm located in the header. These algorithms include "permit overrides" and "first applicable".

While these algorithms work in all cases, when the number of rules increase, the chances of a policy writer forgetting a rule that conflicts with other rules negatively increases. An earlier rule could potentially make all following rules inapplicable if it is general enough. This is the problem area that this thesis is trying to address. Should the writer forget a case, it could give access to the wrong people since permit could

have priority over deny (or possibly worse, deny someone that should have access, such as a technician that is denied access when trying to shut down a plant during a nuclear meltdown).

Policies Policies are a global target and a list of rules. A writer of policies will usually write policies in multiple files to make use of the global target. A request is applied to every policy file to see which action (permit or deny) to take. The following is the syntax for policies:

```
<Policy PolicyId="...Policy's name..."
        RuleCombiningAlgId="...Algorithm to use...">

   <Target>
      ...Syntax above...
   </Target>

   <Rule RuleId="...Rule name..." Effect="...Permit/Deny...">
      ...Syntax above...
   </Rule>
   ...
   <Rule RuleId="...Rule name..." Effect="...Permit/Deny...">
      ...Syntax above...
   </Rule>
</Policy>
```

We can find a sample policy in Appendix E.

3.2 Coq

Coq, a tool developed in France, is built specifically for formal theorem proving. Its name, which is French for rooster, is a play on CoC, a formal language on which Coq is based (more on the CoC later). Coq allows users to define mathematical theorems and helps prove them interactively. The proof can then be checked by the Coq compiler. A theorem to be proven in Coq contains a set of hypotheses, H_1, $H_2, \ldots, H_n$, which we will denote as Hyp in this thesis, and a conclusion, denoted as Con. We write *goal* to denote the theorem: $Hyp \vdash Con$.

Editing Modes Coq has two editing modes, **toplevel mode** and **proof editing mode**. In toplevel, one can define constants, variables, functions, etc., or state lemmas and theorems. Once a lemma or theorem has been stated, Coq changes into proof editing mode. In this mode, we can access tactics to prove theorems (seen later).

General Syntax We give commands to Coq one line at a time. Below is the General syntax for writing code in Coq:

.	Every definition, theorem, tactic, etc. must end with a period.
indent	Identifiers can be any string of letter (case sensitive), numbers, as well as a few special characters such as _ and '. Identifiers must start with a letter.
_	The wild card character in Coq.
|−	Separates the hypotheses from conclusion. $\vdash$ will be used in this thesis to denote Coq's syntax shown to the left.
:=	The assignment operator.
Z, nat	Coq's built-in versions for integers and natural numbers. $\mathbb{Z}$ and $\mathbb{N}$ will be used to denote them.
:	The typing operator. E.g., A : $\mathbb{Z}$ means A is of type $\mathbb{Z}$. It has a similar meaning to $A \in \mathbb{Z}$, thus $\in$ will be used when it applies.
(* *)	Anything found between (* and *) will be commented out (ignored by Coq).
spaces	Consecutive white spaces (spaces and returns) are parsed as one space. Thus we can write long commands on multiple lines and use spaces for alignment.

Propositions **Propositions** are the building blocks of Coq's theorems. Coq implements the calculus of inductive constructions (CIC), which is based on the calculus of constructions (CoC). The CIC is an expressive higher-order logic, and propositions (elements of type Prop in Coq) are the formulas of this logic, i.e. they are objects whose validity can be argued. We do not use the full expressive power of this logic in this thesis, and only present syntax and examples from the sub-logic we use.

We make simple propositions using equalities, inequalities, disequalities, functions and identifiers. E.g., $A + B = 5$, $f(A) > -2$. We can also build propositions from values other than numbers, such as booleans and use-defined types. E.g., $b = true$, $colour(sky) = green$ (propositions don't have to be true). Note that sometimes propositions cannot be proved or disproved. Once we have these simple propositions, we can combine them with the following syntax:

True	A proposition that is always provable.
False	A proposition that has no proof.
Prop	Coq's built-in type for propositions.
->	Implications. E.g., A -> B means A implies B. $\rightarrow$ will be used in this thesis to denote Coq's syntax shown to the left.
/\	Conjunctions. E.g., A /\ B means A and B. $\wedge$ will be used to denote it.
\/	Disjunctions. E.g., A \/ B means A or B. $\vee$ will be used to denote it.
$\sim$	Negation. E.g., $\sim A$ means "not A".
forall	Universal quantifier. E.g., forall x:nat, $x + 5 > 0$ means that for any $x \in \mathbb{N}$, $x + 5 > 0$. $\forall$ will be used to denote it.
exists	Existential quantifier. E.g., exists x:Prop, $P(x)$ means there is a proposition, x, s.t. $P(x)$. $\exists$ will be used to denote it.

Bool The type **bool** is Coq's built-in type for Booleans. They are a basic data type distinct from Coq's propositions. They are defined inductively, and can be used to reason by cases. There are two cases for a bool, say B, $B = true$ and $B = false$. $true$ and $false$ are Coq's built-in notation for these values. We can form equations using bools. E.g., A <b 5 is a bool (and so either true or false, as opposed to $A < 5$, which is a proposition, in which we can't argue by cases). We can also form propositions by

equating bools: A <b $5 = true$. Note that <b (and <=b) is our custom notation for Coq's built-in prefix notation.

Coq also has built-in conditionals for bools:

```
if b then tv else fv
```

where b is a bool. If $b = true$, then this expression has tv as its value, otherwise, it has fv as its value. This statement has the same type as tv and fv, which must be the same.

Definitions (of functions) There are several ways to define things in Coq. The following are the ways used in this thesis:

```
Definition ident: t1 -> ···-> tn -> t := ... .
```

This defines *ident* to be the term to the left of :=. It has n arguments of type $t_1, \ldots, t_n$. It has a return value of type t. E.g.,

```
Definition MAX: Z := 86400.
Definition inRange: Z -> Z -> Z -> bool := ... .
```

The first defines a constant with value 86400. It has no arguments, and returns an integer. The second defines a function called *inRange* (whose definition is omitted for now). It has three integer arguments and returns a bool. In this example, we have to name our arguments if we want to use them in the definition. We do this by using the following syntax after the `:=`.

```
fun a1 a2 ... an =>.
```

This takes the first n non-named arguments, and names them $a_1, \ldots, a_n$ (resp.). E.g.,

```
Definition inRange: Z -> Z -> Z -> bool := fun req m M => ... .
```

We can also name our arguments in the header using this syntax:

```
Definition ident (a1:t1) ... (an:tn) : t := ... .
```

This will have a same effect as the above using $fun\ a_1 \ldots a_n =>$. Should two or more arguments have the same type, we can group them together. Below are the syntax and an example:

```
Definition ident (a11 a12 ... a1m :t1)...(ak1 ... akn:tk) : t := ... .
Definition case1' (m1 M1 m2 M2: Z) : bool := ... .
```

This defines $case1'$. It has four arguments: $m1, M1, m2, M2 \in \mathbb{Z}$ and returns a bool.

We can get an instance of these definitions by writing $ident\ a_1\ a_2 \ldots a_n$, where a_1, ..., a_n are arguments. E.g., if we define:

```
Definition isLessThan (x y: Z) : Prop := x < y.
```

$isLessThan$ 3 5 would simplify to $3 < 5$, which is True. We can omit the last few arguments to get new functions with fewer arguments. E.g., ($isLessThan$ 1) is a function with one argument, y, and returns a Prop with value $1 < y$.

One can't define a recursive function using the keyword Definition. To do so, we need to use the keyword Fixpoint, with syntax:

```
Fixpoint (a1:t1)...(an:tn) {struct ai}: type := ... .
```

Coq only allows us to define recursive functions that terminate. Because of this, it needs to know how it gets simpler in each call. Thus we add the new argument, $\{struct\ a_i\}$, which says that a_i will decrease structurally. Below is an example:

```
Fixpoint listCheck (l1: list srac)(sr2:srac){struct l1} : bool := ... .
```

In this case, we defined a recursive function called $listCheck$, with arguments l_1 and sr_2. $\{struct\ l_1\}$ tells Coq that l_1 will decrease structurally (it's a list that will decrease in size) with every subsequent call. Note that this definition also illustrates Coq lists. $list$ is a "type constructor" that takes a type argument ($srac$ in this case). $list\ srac$ is a new type, whose elements are lists that contain only elements of type $srac$.

Records **Records** provide a way to store many values together. Below are their syntax and an example:

```
Record name: Set := constructorName {a1:t1;...; an:tn}.
```

```
Record rule: Set := ruleCons {acc:access;  subjects: list srac;
resources: list srac; actions: list srac; conditions: list srac}.
```

This definition is used to create rules. Each rule contains an access action (acc : $access$) which has either permit or deny as value (defined below), and the four lists needed in each rule ($subjects$, $resources$, $actions$ and $conditions$). We can access each of these fields using the name of the field we want to access followed by the identifier for the rule. E.g., $acc\ r_1$ returns either permit or deny (whichever value the rule r_1 has).

Inductive types **Inductive types** are defined by providing cases. E.g., bool is an inductive type with two cases, true and false. The cases can be recursive too, e.g., natural numbers has cases 0 and $S(n)$, where n is a natural number and S is the successor function. The general syntax for an inductive set and Coq's built-in type nat, which is defined inductively, as an example is shown below:

```
Inductive ident: Set := case1 |...| casen.
```

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

Usually one writes each case on a new line. A vertical bar is also allowed before $case_1$. Because of this, we can start the first case on a new line starting it with a |. This helps with alignment.

There are many ways to define a case. Below is the form used in this thesis, and some examples:

```
caseName: a1 -> ··· -> an -> ident.
```

```
Inductive access: Set := permit : access | deny : access.
```

ident must be the same as the type being defined. If there are no arguments, it can be omitted. E.g., the following is equivalent to the above example:

```
Inductive access: Set := permit | deny.
```

Below is an example with arguments:

```
Inductive srac : Set :=
| timeInRange : tValue -> tValue -> srac
...
| intGt : iValue -> srac.
```

$tValue$ and $iValue$ will be defined later.

Lemmas and Theorems **Lemmas** and **theorems** are proven propositions in Coq. We can define lemmas and theorems as follows:

```
Lemma ident : P.
Theorem ident : P.
```

where *ident* is the name of our lemma or theorem and P, a proposition, is the theorem itself. In Coq, there is no difference between lemmas and theorems (other than the keyword), thus anywhere that only mentions one of these, the other can also be used in the same way. Once we have defined a theorem, Coq changes into proof editing mode, and we are given goals. We need to prove each of the given goals in order to conclude the theorem. Below is an example of a simple lemma:

```
Lemma Zle_lt_trans: forall A B : Z, A < B -> A <= B.
```

This lemma says that for each pair of integers, A and B, if $A < B$, then we can conclude that $A \leq B$.

The initial goal is usually of the form: $\vdash Con$, with no hypotheses and the conclusion being the whole theorem. We can add a hypothesis, say *ident* which assumes P, before the Lemma command with the following syntax:

```
Hypothesis ident : P.
```

This will add $ident : P$ to Hyp giving the initial goal the form:
$ident : P, ident_1 : P_1, \ldots, ident_n : P_n \vdash Con$, where each $ident_i : P_i$ are hypotheses previously added by the same method. Any theorem following a *Hypothesis* command will have the hypothesis added in Hyp.

Once a lemma or theorem has been defined, it must be proven in order to be used. We do this using the keyword 'Proof.' We then follow up with a series of tactics to prove it. Once proven, we save it using 'Qed.'. This allows the lemma or theorem to be used in other proofs. Below is an example of a proof of the previous lemma.

```
Lemma Zle_lt_trans: forall A B : Z, A < B -> A <= B.
Proof.
intros.
omega.
Qed.
```

Note that *intros* and *omega* are tactics that will be described later.

Tactics When in proof editing mode, we use **tactics** to make progress in proofs. Tactics are commands we can give Coq that are applied in a backwards direction and reduce a goal to a (hopefully) simpler goal. Tactics may split a goal into more than one subgoal, each new subgoal has to be proven to conclude the original goal.

There are many built-in tactics, and we can write some of our own using the Ltac language. Ltacs simply combine existing tactics and will be illustrated later. The following is a list of the built-in tactics used here and a description of what they do, starting with a few common ones and examples.

intro	Replaces a goal: $Hyp \vdash h \to C$ with: $Hyp, h \vdash C$, and a goal: $Hyp \vdash \forall x \in t, C$ with: $Hyp, x \in t \vdash C$. We can add an optional argument to this tactic: intro H. Adding this argument gives the new hypothesis the name H instead of the name generated by Coq.
intros	Is equivalent to using intro repeatedly until nothing can be done. We can add a list of arguments, $H_1\ H_2 \ldots H_n$, to intros. This is equivalent to intro H_1. ... intro H_n.
reflexivity	Proves a goal with a conclusion: $c = c$.

```
Lemma ex1 : forall x:Z, x = x.
Proof.
intro.
reflexivity.
Qed.
```

We start with the goal: $\vdash \forall x \in \mathbb{Z}, x = x$. Applying the tactic intro brings x into the hypothesis by choosing an arbitrary $x \in \mathbb{Z}$. This gives us the following goal: $x \in \mathbb{Z} \vdash x = x$. At this point, we can complete the proof using reflexivity. Once completed, we save the lemma using Qed.

generalize H	H must be a theorem or hypothesis. Replaces Con with $H \to Con$
assumption	Completes any goal where $Con \in Hyp$.

```
Lemma ex2 : 5 = 5.
Proof.
generalize (ex1 5).
intro.
assumption.
Qed.
```

We could have again used reflexivity to solve this lemma, but in this case, we will use our previous lemma, ex_1, to prove this. When a lemma contains a forall $(x_1 : t_1) \ldots (x_n : t_n)$ in the head of a lemma, the variables $x_1, \ldots, x_n$ become arguments to the lemma. We can get a specific instance of the lemma by supplying these arguments (in the same way we used arguments in formulas: the name of the lemma followed by the arguments). Here, x is an argument to the lemma, and so, $ex_1\ 5$ will replace every instance of x by 5 in the lemma, giving us $5 = 5$. Using generalize on this

adds $5 = 5 \rightarrow$ in the conclusion giving us the new goal: $\vdash 5 = 5 \rightarrow 5 = 5$. We can use intro in the same way we did above to get $5 = 5$ into Hyp giving us the goal: $5 = 5 \vdash 5 = 5$. Since $Con \in Hyp$, assumption completes the proof.

```
Lemma ex2_1 : forall (A B C : Prop), (A -> B) -> (B -> C) -> (A -> C).
```

There are several ways to prove this lemma. Of those, we will see four ways. Note that Coq is powerful enough to prove this lemma automatically, but we show these proofs to illustrate basic tactics. The first uses generalize as follows:

```
Proof.
intros A B C HAB HBC HA.
generalize (HAB HA).
assumption.
Qed.
```

We start with intros, giving us the goal:
$A\ B\ C : Prop$, $HAB : A \rightarrow B$, $HBC : B \rightarrow C$, $HA : A \vdash C$. Then we use generalize on $HAB : A \rightarrow B$, in a way similar to its use in the above lemma. In this case, HAB has one argument: A (which we know as HA). generalize ($HAB\ HA$) adds $B \rightarrow$ in the conclusion, giving the new goal:
$A\ B\ C : Prop$, $HAB : A \rightarrow B$, $HBC : B \rightarrow C$, $HA : A \vdash B \rightarrow C$. $Con \in Hyp$, thus assumption completes the proof.

The second way to prove this lemma is with apply:

apply H — H must be a theorem or hypothesis of the form $A_1 \rightarrow \cdots \rightarrow A_n \rightarrow Con$. Creates n subgoals: $Hyp \vdash A_i$. If H has no arguments, it solves the goal directly.

```
Lemma ex2_2 : forall (A B C : Prop), (A -> B) -> (B -> C) -> (A -> C).
Proof.
intros A B C HAB HBC HA.
apply HBC.
apply HAB.
assumption.
Qed.
```

After intros, we get with the same goal as before:

$A\ B\ C : Prop,\ HAB : A \to B,\ HBC : B \to C,\ HA : A \vdash C.$

Using apply HBC gives us this new goal:

$A\ B\ C : Prop,\ HAB : A \to B,\ HBC : B \to C,\ HA : A \vdash B.$

Followed by apply HAB:

$A\ B\ C : Prop,\ HAB : A \to B,\ HBC : B \to C,\ HA : A \vdash A.$

Which is completed with assumption.

Since we already knew A (since $HA : A$), we could have used it as an argument for HAB, giving us B ($HAB\ HA : B$). Thus, apply ($HAB\ HA$) would have finished the proof after applying HBC, saving us one step. Even better, we can also use this proof of B as an argument to HBC to conclude C directly after intro, giving us this proof:

```
Proof.
intros A B C HAB HBC HA.
apply (HBC (HAB HA)).
Qed.
```

The third way is with assert:

assert P Splits the goal into two parts: $Hyp \vdash P$ and $Hyp, P \vdash Con$. This is used to add a new Hypothesis P. We first prove P is true, then can use P to prove our original goal.

```
Lemma ex2_3 : forall (A B C : Prop), (A -> B) -> (B -> C) -> (A -> C).
Proof.
intros A B C HAB HBC HA.
assert B.
apply (HAB HA).
apply (HBC H).
Qed.
```

assert B will split the goal into two subgoals:

$A\ B\ C : Prop,\ HAB : A \to B,\ HBC : B \to C,\ HA : A \vdash B$, and

$A\ B\ C : Prop,\ HAB : A \to B,\ HBC : B \to C,\ HA : A,\ HB : B \vdash C$

We have to prove both of these to conclude the lemma. In this case, both are provable with methods seen above.

The fourth way is with cut:

cut P	Splits the goal into two parts: $Hyp \vdash P \to Con$ and $Hyp \vdash P$. This is used to replace Con with a stronger conclusion, P.

```
Lemma ex2_4 : forall (A B C : Prop), (A -> B) -> (B -> C) -> (A -> C).
Proof.
intros A B C HAB HBC HA.
cut B.
assumption.
apply (HAB HA).
Qed.
```

Similar to assert, cut will split the goal into two subgoals. Unlike assert, cut works in a backwards direction, by changing the goal into something stronger (where the new conclusion implies the old), and then proving this new stronger conclusion.

After using cut B on the above lemma, we get the following subgoals:

$A\ B\ C : Prop$, $HAB : A \to B$, $HBC : B \to C$, $HA : A \vdash B \to C$, and

$A\ B\ C : Prop$, $HAB : A \to B$, $HBC : B \to C$, $HA : A \vdash B$

The first, to prove B is a stronger conclusion than C, then proving B. Again, we have to prove both of these to conclude the lemma, and can do so with previous methods.

destruct H	Decomposes the top most operator in the hypothesis H. E.g., destruct H when $H = H_1 \wedge H_2$, would add H_1 and H_2 to Hyp.
split	When $Con = A \wedge B$, it creates two subgoals: $Hyp \vdash A$ and $Hyp \vdash B$.
T_1;T_2	Applies the tactic T_2 to all subgoals generated after applying T_1.

```
Lemma ex3: forall (A B: Prop), A /\ B -> B /\ A.
Proof.
intros A B H.
destruct H.
split.
assumption.
assumption.
Qed.
```

After intros, we get the following goal: $A\ B : Prop$, $H : A \wedge B \vdash B \wedge A$. Using destruct on H gets us two hypotheses A and B (Coq picks the names for these) forming the following goal: $A\ B : Prop$, $H : A$, $H_0 : B \vdash B \wedge A$. Using split makes us have to prove each half of the conjunction separately:

$A\ B : Prop$, $H : A$, $H_0 : B \vdash B$, and

$A\ B : Prop$, $H : A$, $H_0 : B \vdash A$.

Both of these goals can be completed using assumption.

Since assumption was a proof for both new subgoals, we could have simplified our proof by telling Coq to use assumption on any subgoal created by split. We do this by using ';' instead of '.' after split:

```
Lemma ex3' : forall (A B: Prop), A /\ B -> B /\ A.
Proof.
intros A B H.
destruct H.
split;
assumption.
Qed.
```

destruct H	Destructs the top most operator in H. E.g., when $H = H_1 \vee H_2$, creates two subgoals: $Hyp, H_1 \vdash C$ and $Hyp, H_2 \vdash C$. If $H = \exists x : P(x)$ destruct would instantiate x, giving $P(x_n)$, where n is the lowest unused subscript for x (starting at 0).
left	Replaces Con with the first disjunction. E.g., Using left when Con is $A \vee B$, goal becomes $Hyp \vdash A$.
right	Replaces Con with the second disjunction. E.g., Using right when Con is $A \vee B$ replaces goal with $Hyp \vdash B$.
$T;[T_1 \vert \cdots \vert T_n]$	This expands the functionality of ;. Instead of applying the same tactic to all subgoals, it applies T_i to the ith subgoal.

```
Lemma ex4: forall (A B: Prop), A \/ B -> B \/ A.
Proof.
intros.
destruct H.
right.
assumption.
left.
```

```
assumption.
Qed.
```

After intros, we get the following goal:

$A\ B : Prop,\ H : A \vee B \vdash B \vee A.$

Using destruct on H gets us two subgoals:

$A\ B : Prop,\ H : A \vdash B \vee A$, and

$A\ B : Prop,\ H : B \vdash B \vee A$

Using right on the first gives us the following:

$A\ B : Prop,\ H : A \vdash A$ which we can prove with assumption. The second subgoal is solved similarly. In this example, we can use a more general version of ; to simplify our proof:

```
Lemma ex4': forall (A B: Prop), A \/ B -> B \/ A.
Proof.
intros.
destruct H;
[right | left];
assumption.
Qed.
```

After destruct H, we apply right to the first subgoal and left to the second subgoal. After that we apply assumption to both. The proof tree for this proof is identical to the previous proof, however it looks much cleaner.

Coq has a lot of automated proving tactics. Below are those that are used in this thesis:

trivial	Completes most goals that can be proven easily. In the examples above, trivial could have been used instead of assumption and reflexive.
tauto	Can complete proofs that only contain tautologies. All of the above lemmas could have been entirely proven using tauto.
omega	An expansion to tauto which specializes in goals containing integers.
auto/eauto/ auto with *	three more automated provers which use many built-in theorems to try and solve the current goal.
intuition	Applies a set of specified tactics, making as much progress to the proof as possible (often completing it). Unlike the other auto-provers, that either complete a subgoal entirely or fails/leaves it unchanged, intuition can partially complete a subgoal.

We can increase the power of some auto-provers (such as auto and eauto) using:

```
Hint Resolve lem1 lem2 ... lemn.
```

This will add $lem_1, \ldots, lem_n$ to the list of lemmas that are tried by some auto-solvers.

The following is the rest of the tactics used in the code associated with this thesis:

case_eq H	Splits the goal into n new subgoals, one per case H can take. Each subgoal becomes $Hyp, Hcase_i \vdash Con$. E.g., if H : bool, it would split the goal into two subgoals: $Hyp, H = true \vdash Con$ and $Hyp, H = false \vdash Con$.
clear H	Removes a hypothesis H which is no longer needed from Hyp. We can also use clear -H, which removes all but the hypothesis H, and clear dependent H which clears H and all of its dependents. This makes it easier for provers (real or automated) to complete a goal, as there are less hypotheses to use.
decompose[o] H	o must be a list of operators separated by spaces. It destructs H until the top operator is not in o. E.g., decompose [and or] H with $H = (\sim (H_1 \wedge H_2)) \wedge (H_3 \vee H_4)$, will create two subgoals: $Hyp, \sim (H_1 \wedge H_2), H_3 \vdash Con$ and $Hyp, \sim (H_1 \wedge H_2), H_4 \vdash Con$
do n T	Applies tactic T n times.
exfalso	Replaces Con with False. Used when there is a contradiction in Hyp.
exists x_0	Instantiates an $\exists x$ in Con with x_0.
first $[T_1 \vert \cdots \vert T_n]$	Tries to apply each tactic starting from left until one succeeds.
[1]fold h	Opposite of unfold (see below). Replaces h's definition by h.

induction h	Generates one subgoal for each form h can take. It then rewrites every occurrence of h into the form for that subgoal. For recursive forms, it adds an induction hypothesis to Hyp. E.g., if h : bool, it generates two subgoals: $Hyp, h = true \vdash Con$ and $Hyp, h = false \vdash Con$.
remember x	Creates a copy of x by adding a new hypothesis $x_0 = x$.
rename H into H_2	Changes H's name to H_2. The goal is essentially unchanged.
repeat T	Repeatedly applies a tactic T until it either fails or until it doesn't change the goal.
[1]rewrite H	H must be an equality, say $H : A = B$. Replaces any occurrence of A with B.
[1]rewrite <- H	H must be an equality, say $H : A = B$. Replaces any occurrence of B with A.
set $(id := def)$	Creates a new variable called id, defined to be def. Adds id to Hyp.
simpl	Simplifies Con by unfolding definitions or evaluating terms.
solve $[t]$	Applies the tactic t. It either proves Con entirely or fails. It is useful with the try tactic (see below): try (solve $[t]$), since either it proves the goal completely or leaves it unchanged. It is a quick way to eliminate easily proven subgoals that have the same proof.
try T	Applies a tactic T, should it fail, it does nothing, otherwise it's equivalent to just using T. This tactic never fails.
[1]unfold h	Replaces h by its definition.

[1]These tactics are used on Con by default. They can be used on a hypothesis H instead by adding "in H" at the end of the tactic (e.g., fold h in H) or used anywhere that applies with "in *".

Imports and exports Coq has many built-in theorems, all of which are pre-proven in libraries. To use these, we have to **import** the library they are in. To import a library, say *ident*, which contains the theorems we want to use, we use the syntax:

```
Require Import ident.
```

We can also import our own theorems with the same syntax, where *ident* needs to be the name of the file containing these theorems. If both the imported and importing file require the same library, we use:

```
Require Export ident.
```

in the imported file. This makes the library *ident* available to both files with only one command.

Sections We can split up Coq files into **sections**. We do this using the following commands to start a section called *ident*, and to end it:

```
Section ident.
End ident.
```

Once inside a section, we can define hypotheses and variables that will become arguments when using the theorems outside of the section. An example is shown below:

```
Section exSect.
Variables x y :nat.
Hypothesis exHyp: y = S x.
Lemma exLem: y > x.
Proof.
rewrite exHyp; auto.
Qed.

Check exLem.
End exSect.

Check exLem.
```

In this example, we declared that x and y were going to be natural numbers for the remainder of the section. Then, we assume that $y = S\ x$ (recall that S is Coq's built-in successor function, i.e. $S\ x = x + 1$). This hypothesis will also stay until the end of the section. We then prove that these three facts imply that $y > x$. `Check exLem.` allows us to see what the theorem is. When we check inside the section, we see: `exLem : y > x`. Outside of the section, we see: `exLem : forall x y : nat, y = S x -> y > x`. Notice that *exLem* doesn't need any arguments inside the section, and we can apply it directly. Outside of the section however, we need to supply three arguments. Those arguments being, two *nat* arguments (x and y) and a proof that $y = S\ x$. Should we try to use the lemma in an importing file, we would also need to supply these three arguments.

Match **match** is used to decide which case we are in and assign a corresponding result for that case. Below is the syntax for it and an example:

```
match H with case1 => result1 | ... | casen => resultn end.
```

```
Definition rank : srac -> Z :=
fun sr =>
match sr with
| any => 0
| timeInRange _ _ => 1
| timeGe _ => 2
| timeLt _ => 3
| intGt _ => 4
end.
```

This assigns ranks (integers) to the argument *sr* of type *srac*. Recall that _ is a wild card character. By using it here, we are ignoring the arguments of the functions and just looking at its identifier.

When in proof editing mode, match can act as a tactic. We do this by having a tactic as its result. Any variables must be preceded by a '?', after which, they can be used in the result. It provides a convenient way to extract variables from equations. Below is an example.

```
match H with
if ?a then _ else _ => test a
end.
```

In this example, we see if H is an if statement, and use the tactic test on the condition in the if statement. We can also match our hypotheses using this syntax:

```
match goal with
  | [ x : h |- _] => T x
end.
```

This will look for the first hypothesis, x, that matches h, and apply the tactic T to it. E.g.

```
match goal with
| [ x : _ = _ |- _ ] => rewrite x
end.
```

will search for a hypothesis that is an equality, and use the tactic rewrite on it. x can be any identifier, or _ if we don't intend on using it in the result. 'goal' however is a keyword in Coq denoting the current goal.

Similarly, we can match our conclusion with:

```
match goal with
  | [ |- c] => T
end.
```

This will try to match Con with c. If it does, then it applies tactic T.

Another useful match is to match types. We do this using:

```
match type of H with ...
```

Since hypothesis statements are types, this is useful when trying to match something within a hypothesis.

Operators It is possible to create custom operators in Coq. We do this using the following syntax:

```
Infix "op" := func (at level assoc): scope.
```

This will define *op* (it must be surrounded by quotes), which is the symbol for the new operator (e.g. *op* could be $<$). This operator will be an infix version of the pre-existing function $func$. $assoc$ is an integer which will define its associativity with other operators. A smaller number means that operator is used first. E.g. $+$ has a lower associativity level than $=$, and thus $A + 2 = 7$ will be taken as $(A + 2) = 7$ and not $A + (2 = 7)$. We then add a scope to the operator (more on this later). The following is an example of a custom operator:

```
Infix "<b" := Zle_bool (at level 70): Z_scope.
```

After this definition, Coq will consider <b and *Zlt_bool* as equivalent, with the exception of <b being an infix operator and *Zlt_bool* a prefix operator. E.g. A <b B and *Zlt_bool* A B will be equivalent. We set its associativity level to 70 (which is the same level as the other inequalities).

We can assign more than one function to each operator. An example is with <: $A < B$ acts differently if A and B are natural numbers than when they are integers (the two types are defined differently and thus < acts accordingly). Because of this, we need to ensure the right version is used. This is where *scope* comes in. In the above example, we defined <b to work in *Z_scope*. If the current scope is anything but *Z_scope*, the operator will not work. We can change the scope in two ways: by changing the default scope or by telling Coq to use a different scope once. We can change the default to *Z_scope* with the following syntax:

```
Open Scope Z_scope.
```

After this command, we can now use our new operator. We can open more than one scope at a time. When using an operator that is defined in more than one scope, it prioritizes function in the most recently opened scope. Thus opening *nat_scope* then *Z_scope* will result in a priority to the functions in *Z_scope*.

The second way is to just change the scope for the one operation. We do this by adding `%key` after the operation. Each scope is defined with a *key*. The *key* for *Z_scope* is *Z*. We could have used <b by using (A <b B)%*Z* instead of opening the scope. In this thesis, we only used the scopes *nat_scope* with $key = nat$ and *Z_scope* with $key = Z$.

As stated before, we don't use Coq's full power. For readers who wish to learn more about it, they are welcome to look at Coq's reference manual [15] or some read some books written about it such as the one by Adam Chlipala [4].

Chapter 4

Implementation

This chapter will show how we encode XACML into Coq. In Section 4.1 we will encode rules, define requests and define what it means for two rules to conflict based on our encoded definitions. Two rules conflict if there is a request that is both permitted and denied by the policy (ignoring the rule combining algorithm such as first applicable). The conflict detection algorithm uses the notions of intersections which captures the set of requests that both rules apply to. Section 4.2 presents the Coq development for expressing intersections between rules. Section 4.3 will use these intersections to define the conflict detection algorithm.

We have defined a naming convention for this thesis as follows:

m	This will represent an argument for a lower bound. E.g. for a time greater than or equal to five, we use $timeGe\ m$, with $m = 5$.
M	This will represent an argument for an upper bound. For time ranges, we will use M' as the original upper bound, and M as the converted bound (more on this later).
sr	These will represent one of the elements of type *srac* contained in a rule.
1, 2	These will follow the above to help distinguish which rule they come from. When working with two rules, every variable associated with the first rule will be followed by a one, and similarly, every variable associated with the second rule will be followed by a two. E.g. m_1 is a lower bound in the first rule and M_2 is an upper bound in the second rule.
req	This will represent a request argument. E.g. if a request was made at 10am, then $req = 10am$. Note that req is an integer, and $10am$ would be stored as 36000 (conversions shown later).
reqv	Similar to req, but this variable also contains the kind of request (either time or integer). E.g. in the example above, $reqv = timeValue$ 36000.
functions	This will be used to refer to Coq functions. E.g. $M'ToM$ (defined later) will be referred to as a function.
srac	This will be used to refer to XACML functions. E.g. $timeInRange$ will be referred to as an *srac*. See discussion after the definition of inductive type *srac* on page 34

We will explain what functions do after they are defined.

4.1 Representing XACML in Coq

This section will explain how XACML syntax is encoded in Coq. We will run through lines from the Coq implementation and explain how they are used.

```
Inductive access: Set := permit | deny.
```

Every rule either permits or denies access. This type will let us differentiate between a rule that permits and a rule that denies access.

```
Inductive tValue: Set := timeValue : Z -> tValue.
Inductive iValue: Set := intValue : Z -> iValue.
```

$tValue$ and $iValue$ will be used as XACML types in building elements of type *srac* as well as types for each kind of request. We can get instances of these types by writing the constructor name ($timeValue$ or $intValue$) followed by its argument (an integer in both cases).

A $timeValue$ is represented as an integer, where the number is the amount of seconds after midnight. In XACML, time is represented as a triple of integers: $H : M : S$, where H, M and S are the number of hours, minutes and seconds past midnight, respectively. These values must be positive, with H less than 24 and both M and S less than 60. We can convert between these two versions of time values. We get the single integer version, say t, as follows: $t = 3600H + 60M + S$ and the triple: $S \equiv t (\text{mod } 60)$, $M \equiv \frac{t-S}{60} (\text{mod } 60)$ and $H = \frac{t-60M-S}{3600}$.

```
Definition MAX := 86400.

Hypothesis timeConst: forall tv : tValue,
match tv with
  | timeValue t => 0 <= t < MAX
end.
```

Each value in the $timeValue$ triple is bounded. Because of this, the resulting integer, t, is also bounded. We add this fact as a hypothesis, that is, any $timeValue$ is bounded between 0 and $MAX = 24 \times 60 \times 60$, the number of seconds in a day. All of the proofs only use the fact that MAX is a positive number, and not the actual value, thus we could change MAX to any other value to get arbitrary precision for the time values.

```
Inductive reqValue: Set :=
| timeReq : tValue -> reqValue
| intReq : iValue -> reqValue.
```

This will be used to construct requests. Note that we only need a constructor for the argument types that will be used in requests (in this case all of them).

```
Inductive srac : Set :=
| any : srac
| timeInRange : tValue -> tValue -> srac
| timeGe : tValue -> srac
| timeLt : tValue -> srac
| intGt : iValue -> srac.
```

srac stands for subjects, resources, actions or conditions. *srac* is a type whose constructors are XACML functions. However, in this thesis, we will use *srac* to mean "*srac* constructor", that is, *srac* will stand for one of the five XACML functions (*any*, *timeInRange*, *timeGe*, *timeLt* or *intGt*). *sracs* build XACML formulas to test against to see if a rule applies. We have defined five *srac*: *any* with no arguments, *timeInRange*, with two time arguments, *timeGe* and *timeLt*, each with one time argument, and *intGt* with one integer argument. We will define what the *srac* express, and what type of request they apply to later.

```
Record rule: Set := ruleCons {
  acc:access;
  subjects: list srac;
  resources: list srac;
  actions: list srac;
  conditions: list srac
}.
```

This definition is used to create rules. Each rule contains an access action (acc : access) which takes either permit or deny as value, and the four lists needed in each rule (subjects, resources, actions and conditions). As opposed to XACML, subjects, resources and actions are not joined together in a target. Also, we ignore the global target. We can take the intersection of the global target with the target in the rule to obtain a combined target. Since a central purpose of this thesis is to take intersections, we can run the intersections algorithm on the global target and rule's target to get the combined target.

```
Record request: Set := requestCons {
```

```
 subj : reqValue;
 res: reqValue;
 acti: reqValue;
 cond: reqValue
}.
```

This definition is used to create requests. We can get an instance of a request by putting the name of its constructor (*requestCons*) followed by each argument in the order they appear. E.g. *requestCons s r a c* will form a request with subject s, resource r, action a and condition c. Unlike XACML, which uses a list of values for each field to denote multiple properties of the single request, we will only use one value. The reason we limit requests is that if we allow multiple values per field, every rule which applies to at least one request (usually all of them, otherwise it would be a useless rule) would intersect with all other rules trivially. E.g. a request with subject $s = [s_1, s_2]$ where one rule applies to s_1, and a second rule applies to s_2, then s is a trivial intersection (for the subject field). E.g. a first rule whose subject applies to anyone called Mark and another rule to anyone called Tom. If we choose s = [Mark, Tom] (one user making a request using two names), then both rules apply to this subject, and so s is an intersection. This thesis will only find non trivial intersections, where one single request value (one per field) matches both rules. Should an *srac* need more than one value for it to apply, a request type can be made which contains both values. This restriction still allows any function to be implemented, it only eliminates the trivial conflicts.

The following functions all relate to time. They will be used to see if a rule applies to a given request.

```
Definition inRange': Z -> Z -> Z -> bool :=
fun req m M' =>
if (m <=b M') then (m <=b req) && (req <=b M')
              else (req <=b M') || (m <=b req).
```

This function has three arguments, *req*, m and M', which are the request value, the lower bound of the range and the upper bound of the range, respectively. The

function sees if req is in the range m to M'. If $m \leq M'$ then the function returns true if and only if $m \leq req \leq M'$. If $m > M'$ then the range loops past MAX to 0, and so req must be on the exterior to be considered in the range. E.g. 14 to 5 is the range $[0, 5] \bigcup [14, MAX)$.

```
Definition inRange: Z -> Z -> Z -> bool :=
fun req m M =>
if (M <b MAX) then inRange' req m M
              else inRange' req m (M-MAX).
```

This function is similar to the above, however, we only apply it when $m \leq M$. We ensure this with the following definition for converting M'. We will then use the result of $M'ToM$, stored in the variable M as the new argument for $inRange$.

```
Definition M'ToM : Z -> Z -> Z :=
fun m M' => if (m <=b M') then M' else M' + MAX.
```

This is a function that returns the smallest time, M, that is equivalent to M', such that $m \leq M$. By equivalent, we mean that the same amount of time has past after midnight, i.e., either $M = M'$ if $m \leq M'$ or $M = M' + MAX$ otherwise (the second will always give the desired result if $M' < m$ as both values are less than MAX). This simplifies the time ranges as we only need to test $m <= req <= M$. The two versions ($inRange'$ and $inRange$) are proven to be equivalent.

The function found in Figure 4.1 is where we define how each $srac$ is evaluated, which depends on the form of the request. E.g. $timeInRange$ is in the $timeReq$ block, thus any request, $reqv$, must have type $tValue$ to apply. If the request is of the right type, then it uses $inRange$ (defined above) to see if the $srac$ applies to the request. any is found in both type blocks, thus can apply to either a time or integer request.

The other three functions act similarly, only differing in the types and functions. Zge, Zlt and Zgt are three built-in functions that tests if the first argument

```
Definition valueMatch: reqValue -> srac -> Prop :=
fun reqv sr =>
match reqv with
|  timeReq (timeValue req) =>
   match sr with
   | any => True
   | timeInRange (timeValue m) (timeValue M) => inRange req m M = true
   | timeGe (timeValue m) => Zge req m
   | timeLt (timeValue M) => Zlt req M
   | _ => False
   end

|  intReq (intValue req) =>
   match sr with
   | any => True
   | intGt (intValue m) => Zgt req m
   | _ => False
   end
end.
```

Figure 4.1: Definition of $valueMatch$

is greater than or equal, less than, and greater than or equal (respectively) to the second argument.

```
Fixpoint listMatch (reqv: reqValue)(ls:list srac){struct ls} : Prop :=
  match ls with
  | nil => False
  | cons sr ls' => (valueMatch reqv sr) \/ (listMatch reqv ls')
  end.
```

This function tests if a request, $reqv$ matches a list of $srac$, ls. For a request to match the list, it only needs to match one entry in the list. We test each entry in the list until either we find a match, or we get to the end of the list ($ls = nil$), in which case the request doesn't apply.

```
Definition rule_permit: request -> rule -> Prop :=
fun rq rl =>
  acc rl = permit /\
  listMatch (subj rq) (subjects rl) /\
  listMatch (res rq) (resources rl) /\
  listMatch (acti rq) (actions rl) /\
  listMatch (cond rq) (conditions rl).

Definition rule_deny: request -> rule -> Prop :=
fun rq rl =>
  acc rl = deny /\
  listMatch (subj rq) (subjects rl) /\
  listMatch (res rq) (resources rl) /\
  listMatch (acti rq) (actions rl) /\
  listMatch (cond rq) (conditions rl).
```

These two functions determine if rq, a request, is permitted or denied (respectively) by rl, a rule. The functions are true if and only if the rule has the correct access action and the rule applies (there is a match on all four components). If $rule\_permit\ rq\ rl$ is true for any rule in the set of rules, then the request (rq) should be permitted. Similarly, if $rule\_deny\ rq\ rl$ is true for any rule in the set of rules, then the request (rq) should be denied.

```
Definition rule_conflict: rule -> rule -> Prop := fun r1 r2 =>
  exists rq:request, (rule_permit rq r1 /\ rule_deny rq r2) \/
                     (rule_deny rq r1 /\ rule_permit rq r2).
```

Should both *rule_permit rq* r_1 and *rule_deny rq* r_2 be true for any request, rq, and two (different) rules, r_1 and r_2, then there is a conflict between r_1 and r_2.

4.2 Intersection of *srac*

In this section, we will discuss how we find the intersections between *srac*. By intersections, we mean that there is a request that applies to both *srac*. E.g. if we have two *srac*, $sr_1 = timeLt\ 8$, which applies to all time values less than eight, and $sr_2 = timeGe\ 3$, which applies to all time values greater than or equal to three. Then the request, $req = 4$ applies to both and is thus an intersection.

Firstly, any two *srac* that take a request of a different XACML type will not intersect (Recall that we consider only XACML types time and integer). Since we only allow requests with one value (which can only be of a single type), these *srac* do not have an intersection. E.g. *timeGe* and *intGt* do not intersect, since *timeGe* needs a request with type time, and *intGt* needs a request of integer type.

The partially applied *srac* *timeGe* m_1 and *timeGe* m_2 intersect for all values of m_1 and m_2. This is because if m is the maximum of m_1 and m_2, then any value that satisfies *timeGe* m is an intersection of the two *srac*. E.g. The intersection between *timeGe* 5 and *timeGe* 7 is every value that satisfies *timeGe* 7. The intersection of *intGt* m_1 and *intGt* m_2 work similarly, as does the intersection of *timeLt* M_1 and *timeLt* M_2, except that the minimum of M_1 and M_2 is used.

An exception for *timeLt* is when $M = 0$. Since no time values are less than 0, it doesn't allow any request. Even though the intersection is empty, we will consider it as "every value less than 0". We do this because it is an empty rule which should be reported to be changed (either fixed if it was supposed to have a use, or removed because it doesn't do anything). This will keep the algorithm simpler and more useful as we don't have to add a special case for 0 but report the empty rule.

timeGe m_1 and *timeLt* M_2 will intersect if $m_1 < M_2$. The intersection in this case is every value between the arguments. E.g. the intersection between *timeGe* 5 and *timeLt* 7 is every number in the range $[5, 7)$.

timeInRange m_1 M_1' will intersect with *timeGe* m_2 in three cases. For these

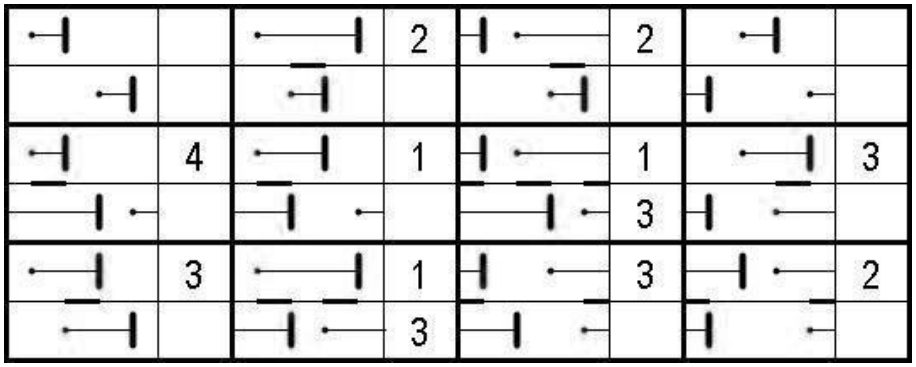

Figure 4.2: Intersection of time ranges for each ordering of arguments

cases, we define m to be the maximum of m_1 and m_2.

1. If $m_2 \leq M_1' < m_1$ the intersection is $[m_2, M_1'] \bigcup [m_1, MAX)$.
2. If $M_1' < m_1$ and $M_1' < m_2$ the intersection is $[m, MAX)$.
3. If $m \leq M_1'$ then the intersection is $[m, M_1']$.

We can also use the contrapositive to see if the pair of *srac* intersect, i.e. if the three cases are false, which only happens when $m_1 \leq M_1' < m_2$, there is no intersection, otherwise there is.

timeInRange m_1 M_1' will intersect with *timeLt* M_2 in a similar way as the above. In this case, it there will be no intersection if $M_2 \leq m_1 \leq M_1'$, otherwise there is.

timeInRange m_1 M_1' will intersect with *timeInRange* m_2 M_2' in a more complicated way. Wlog, we will assume $m_1 \leq m_2$. To find if two time ranges intersect, we use the chart in Figure 4.2. This chart is separated into 12 parts (by the thick lines), each of which separated into four (by the narrow lines). In each part (of 12), the upper left box represents the first range, in which the dot represents m_1 and the bar represents M_1'. Similarly, the lower left box contains m_2 and M_2'. Each of the 12 sections represents a particular ordering on these four values. This ordering is shown by the order of the dots and bars. The closer they are to the left side, the smaller their value. E.g. in the first part, we have $m_1 \leq M_1' < m_2 \leq M_2'$ (one equality is strict, we will see why later). The thicker bars between the ranges represent the intersection

between the ranges. In the right half of each part, we have numbers (or lack of one). These numbers represent how the ranges intersect, defined by the following cases.

```
Definition case1: bool := (m1 <=b M2 - MAX) && (M2 - MAX <=b M1) &&
(inRange req m1 M2).

Definition case2: bool := (m1 <=b M2) && (M2 <=b M1) && (m1 <=b m2) &&
(inRange req m2 M2).

Definition case3: bool := (m1 <=b m2) && (m2 <=b M1) && (M1 <=b M2) &&
(inRange req m2 M1).

Definition case4: bool := (M1 <=b M2 - MAX) && (inRange req m1 M1).
```

These defines the four cases (corresponding to the case number in the chart). Should there be a number in the right half of any part of the chart, then the corresponding case is true (for that particular ordering), otherwise the case is false. Each case has a range in its definition which corresponds to the intersection between the two functions. E.g. $case1$ is true if and only if we are in the three parts with ones in the chart, and in those three parts, the range $[m_1, M_2']$ (the range in the last conjunction) is an intersection of $timeInRange\ m_1\ M_1'$ and $timeInRange\ m_2\ M_2'$.

Sometimes, when arguments are equal, the ordering falls into more than one part of the chart. E.g. if $m_1 = M_1' < m_2 < M_2'$ then we fall into two entries on the table (the (1, 1) entry and the (1, 3) entry). The (1, 1) entry says there is no intersection, but the (3, 1) entry says there is one. Which do we choose? There are three rules for choosing the right entry in the chart:

If the two equal arguments are from the same range, i.e. $m_1 = M_1'$ or $m_2 = M_2'$ then we choose the entry with $m < M$. We choose this entry because $m = M'$ acts in a similar way as $m < M'$ (both accept requests in the range $[m, M']$) while $M' < m$ acts differently (accepting requests in the range $[0, M'] \bigcup [m, MAX)$). In the above example, we would choose the (1, 1) entry.

If the first rule doesn't apply, then we choose the entry with the most intersection

cases (the numbers). E.g. $m_1 < M_1' = m_2 < M_2'$ falls into entry (1, 1) and into entry (3, 1). Entry (3, 1) has one intersection case while entry (1, 1) has none, thus we choose entry (3, 1).
If neither of the above rules apply, then the two entries are equivalent.

Applying these rules to each equality, we get the following for choosing an entry in the table:

$m_1 \leq M_1 < m_2 \leq M_2$	$m_1 \leq m_2 \leq M_2 \leq M_1$		
$m_1 \leq M_1 \leq M_2 < m_2$	$m_1 \leq M_2 \leq M_1 < m_2$		
$m_1 \leq m_2 \leq M_1 \leq M_2$	$m_1 \leq M_2 < m_2 \leq M_1$		
		$M_1 < m_1 \leq m_2 \leq M_2$	$M_2 < m_1 \leq M_1 < m_2$
		$M_1 < m_1 \leq M_2 < m_2$	$M_2 < m_1 \leq m_2 \leq M_1$
		$M_1 \leq M_2 < m_1 \leq m_2$	$M_2 \leq M_1 < m_1 \leq m_2$

Table 4.1: Ranges that define each section in Figure 4.2 for two *timeInRange srac*

```
Definition case1' (m1 M1 m2 M2: Z): bool :=
(m1 <=b M2 - MAX) && (M2 - MAX <=b M1).

Definition case2' (m1 M1 m2 M2: Z): bool :=
(m1 <=b M2) && (M2 <=b M1) && (m1 <=b m2).

Definition case3' (m1 M1 m2 M2: Z): bool :=
(m1 <=b m2) && (m2 <=b M1) && (M1 <=b M2).

Definition case4' (m1 M1 m2 M2: Z):  bool := M1 <=b M2 - MAX.

Definition allCases (m1 M1 m2 M2: Z): bool := (case1' m1 M1 m2 M2) ||
(case2' m1 M1 m2 M2) || (case3' m1 M1 m2 M2) ||(case4' m1 M1 m2 M2).
```

Figure 4.3: four cases for intersections of two time ranges

The four case definitions in Figure 4.3 are similar to the previous four cases (the ones without primes, defined earlier), differing only in that they don't include the intersections appearing as the last conjunction in the earlier definitions. *allCases*

combines the four cases and will be used to test if the ranges $m_1 - M_1$ and $m_2 - M_2$ intersect. If $allCases\ m_1\ M_1\ m_2\ M_2$ is true, then there is an intersection between the ranges.

4.3 Conflict Detection

In this section, we will explain the implementation of the conflict detection algorithm.

The function in Figure 4.4 uses the previously seen functions to test for conflicts between two *srac*. We start out by matching the two *srac* with their corresponding type, then apply the appropriate method to see if there is an intersection. E.g. If $sr1$ and $sr2$ are both $timeInRange$ then we use $allCases$ to see if they intersect.

```
Fixpoint listCheck  (l1: list srac)(sr2:srac){struct l1} : bool :=
  match l1 with
  | nil => false
  | cons sr1 l1' => (sracCheck sr1 sr2) || (listCheck l1' sr2)
  end.

Fixpoint listListCheck  (l1 l2:list srac){struct l2}  : bool :=
  match l2 with
  | nil => false
  | cons sr2 l2' => (listCheck l1 sr2) || (listListCheck l1 l2')
  end.
```

In rules, *sracs* come in lists, where only one *srac* in the list needs to apply for the whole list to apply. We recursively go through each list, calling $sracCheck$ on pairs of elements in the lists to test for intersections on the elements, until either we find an intersection or we reach the end of the list. Since each field in a rule is a list of *srac*, we can call this function on each field to test for intersections in rules.

```
Definition accessDiff: access -> access -> bool :=
fun a1 a2 =>
match a1, a2 with
| permit, deny => true
| deny, permit  => true
| _, _      => false
end.
```

Above is a function to test if the access actions are different.

```
Definition sracCheck: srac -> srac -> bool := fun sr1 sr2 =>
match sr1 with
| any => true
| timeInRange (timeValue m1) (timeValue M1') =>
  match sr2 with
  | any => true
  | timeInRange (timeValue m2) (timeValue M2') =>
    if (m1 <=b m2) then allCases m1 (M'ToM m1 M1') m2 (M'ToM m2 M2')
                   else allCases m2 (M'ToM m2 M2') m1 (M'ToM m1 M1')
  | timeGe (timeValue m2) =>
    if (m1 <=b M1') && (M1' <b m2) then false else true
  | timeLt (timeValue M2) =>
    if (M2 <=b m1) && (m1 <=b M1') then false else true
  | _ => false
  end
| timeGe (timeValue m1) =>
  match sr2 with
  | any => true
  | timeInRange (timeValue m2) (timeValue M2') =>
    if (m2 <=b M2') && (M2' <b m1) then false else true
  | timeGe (timeValue m2) => true
  | timeLt (timeValue M2) => m1 <b M2
  | _ => false
  end
| timeLt (timeValue M1) =>
  match sr2 with
  | any => true
  | timeInRange (timeValue m2) (timeValue M2') =>
    if (M1 <=b m2) && (m2 <=b M2') then false else true
  | timeGe (timeValue m2) => m2 <b M1
  | timeLt (timeValue M2) => true
  | _ => false
  end

| intGt (intValue m1) =>
  match sr2 with
  | any => true
  | intGt (intValue m2) => true
  | _ => false
  end
end.
```

Figure 4.4: Definition of *sracCheck*

```
Definition conflict_check: rule -> rule -> bool :=
fun r1 r2 =>
match r1 with (ruleCons a1 s1 r1 ac1 c1) =>
match r2 with (ruleCons a2 s2 r2 ac2 c2) =>
  accessDiff a1 a2 &&
  listListCheck s1 s2 &&
  listListCheck r1 r2 &&
  listListCheck ac1 ac2 &&
  listListCheck c1 c2
end
end.
```

This function takes two rules and checks for a conflict. For the two rules (r_1 and r_2) to conflict, their access action (a_1 and a_2) must be different and all four of their lists of $srac$ (s_1, r_1, ac_1, c_1, s_2, r_2, ac_2 and c_2) must intersect. This could easily be changed to find redundancies in rules (instead of conflicts) by testing if the access actions are the same instead of testing if they are different.

Now that we can detect if two rules conflict, we can go through rule sets to find all conflicts within a policy file. This part of the implementation (the remainder of the section) was taken from another paper [3]. Even though the paper was finding conflicts in a different setting (in Cisco firewall rules instead of XACML rules), we can use the same algorithms, and as we will see later, the same proof commands completes the proofs for correctness of these algorithms.

```
Fixpoint rrs_conflicts (i n:nat)(r:rule)(rs:rule_set){struct rs}:
  list (nat*nat) :=
match rs with
| nil => nil
| (cons rh rt) => if (conflict_check r rh)
                  then (i,n)::(rrs_conflicts i (S n) r rt)
                  else (rrs_conflicts i (S n) r rt)
end.
```

This function recursively finds conflicts between the ith rule in rs, and the rules with index greater or equal to n (increasing on n). It returns a list of pairs of integers denoting the indices of the rules that conflict (i and all n that conflict).

```
Fixpoint conflicts_aux (n:nat)(rs:rule_set){struct rs}:
  list (nat*nat) :=
match rs with
| nil => nil
| (cons r rs') => (rrs_conflicts n (S n) r rs') ++
                  (conflicts_aux (S n) rs')
end.
```

This function recursively finds all of the conflicts starting at the nth rule. We call the previous function with arguments n and $S\ n$ (Coq's notation for n+1). This finds all conflicts between the n'th rule and every rule after it. We don't call the previous function with arguments n and 0 to avoid finding every conflict twice. We recursively increase n, concatenating (++ is Coq's list concatenating operator) the lists it finds for that particular n.

```
Definition find_conflicts: rule_set -> list (nat*nat) :=
conflicts_aux 0.
```

This function finds all of the conflicts in a rule set.

Chapter 5

Algorithm Correctness

In this chapter, we will prove the correctness, i.e. the soundness and completeness of the conflict detection algorithm. To do this, we will use some custom tactics found in Section 5.1.

5.1 Custom tactics

In this section, we will discuss some of the tactics we created to help simplify proofs. To create a custom tactic, we use the following syntax:

```
Ltac ident a1 a2 ... an := t1; t2; ...; tm.
```

Where $a_1, \ldots, a_n$ are arguments (e.g. hypotheses or identifiers used in the tactics) and $t_1 \ldots, t_m$ are already existing tactics to be applied in succession. When in proof editing mode, writing *indent* $a_1, \ldots, a_n$ will be equivalent to writing $t_1; \ldots; t_m$.

```
Ltac decompAndb H :=
generalize (andbSplit _ _ H); intro temp; destruct temp; clear H.
```

This tactic splits a conjunction of *bool*s into two separate hypotheses. It first uses a lemma which states that

```
forall b1 b2:bool, (b1 && b2) = true -> b1 = true /\ b2 = true.
```

(that $\wedge$ can be used instead of `&&`). It then uses destruct to split the newly formed conjunction. Every Ltac will have a similar look, stringing existing tactics to simplify tasks. As many of the definitions are long, the rest will be omitted from this section, and can be found in the appendix.

```
Ltac caseSimpl b := ...
```

This tactic is an extension to *case_eq* which splits up b into cases (e.g. $b = true$ or $b = false$ for bools), generating multiple goals, one per case. *caseSimpl* also simplifies any occurrences of b in the corresponding *goal* (e.g. it would replace every occurrence of b by *true* in the first subgoal if b was a *bool*).

```
Ltac rewriteSimpl H := ...
```

This tactic works with a hypothesis of type $A = B$. It replaces every occurrence of A with B. It then does some simple arithmetic simplifications (e.g. $a + b - b$ simplifies to a).

```
Ltac test H := ...
```

This tactic tests an inequality H. If the inequality is true or false based on Hyp, it simplifies H to either true or false. If it is unknown, the tactic fails. If H is a conjunction or a disjunction, the tactic tests both operands.

```
Ltac testAll := ...
```

This tactic finds hypotheses that *test* can potentially simplify, and calls *test* when it finds one.

```
Ltac b2PH := ...
```

$b2PH$ is short for boolean to prop in hypothesis. It is used to change proofs containing booleans into an equivalent one with propositions. We do this because many commands (e.g. omega, left and right) can only be used on propositions. It will sometimes split the proof into several parts (when there are disjunctions in hypotheses). With the exception of sometimes splitting the proof into multiple subgoals, this command essentially doesn't change the proof, it just allows other commands to be used.

```
Ltac b2P := ...
```

This is similar to $b2PH$, however, this one converts booleans found in both the hypotheses and in the conclusion. It also splits the current subgoal into several subgoals when there are conjunctions in the conclusion.

```
Ltac simplOrIf := ...
```

This tactic searches for if statements. When it finds one, it uses *test* on the condition in the statement. Should that fail, it splits the proof into two parts, one with the condition true, one with it false. The tactic will also do the same when a bool is in the first disjunction of the conclusion. E.g. if Con is $b = true \vee P$, it will argue by cases on b.

```
Ltac autoSimpl := ...
```

This will automatically use the tactics above (as well as a few built-in auto provers) to reduce the goal as much as possible, often completing it.

The following is a simple example that shows how *autoSimpl* works, by going through all the steps it would take. After each proof command, Coq displays the current goal. The goal after each step will be shown to see what happens in each step. In the display, the hypotheses are above the horizontal line, each hypothesis on its own line. The conclusion is found below the line. Next to the line, the pair (current subgoal/number of subgoals) is displayed.

```
Lemma exLem : forall x y z: Z,
(if x <b y then y <b z else x <b z) = true -> x < z.
Proof.

______________________________________(1/1)
forall x y z : Z, (if x <b y then y <b z else x <b z) = true -> x < z
```

As said previously, the goal usually starts with everything in the conclusion. So we'll use *intros* to get things started.

```
intros.

x y z : Z
H : (if x <b y then y <b z else x <b z) = true
______________________________________(1/1)
x < z
```

First, *autoSimpl* will try to simplify inequalities. We see $x <_b y$. Let's try to simplify that.

```
test (x <b y).

Error: No matching clauses for match.
```

So the tactic didn't work. Coq will not let us use it. We'll have to backtrack and argue by cases.

```
caseSimpl (x <b y).

x y z : Z
H : (y <b z) = true
C : (x <b y) = true
______________________________________(1/2)
x < z


______________________________________(2/2)
x < z
```

This tactic split the goal into two different subgoals. Note that we could have used the tactic *simplOrIf* to do the separation into cases. *simplOrIf* would have found the *if* statement and called the tactic *caseSimpl* as we did.

Coq only displays the hypotheses for the first goal, so let's work on that one. It looks like we can use transitivity on those two hypotheses to conclude the goal, but as stated, many commands don't work with booleans. So let's change those hypotheses into something more useful.

```
b2PH.
```

```
x y z : Z
C : x < y
H : y < z
______________________________________(1/2)
x < z


______________________________________(2/2)
x < z
```

Now we can use transitivity to complete the subgoal. *omega* will do just that (and a lot more if needed).

```
omega.
```

```
x y z : Z
H : (x <b z) = true
C : (x <b y) = false
______________________________________(1/1)
x < z
```

That completed the first subgoal, so Coq moves on to the second. In this case, the conclusion is one of our hypotheses. The only problem is that it is a boolean. We will use the same trick as last time.

```
b2PH.
```

```
x y z : Z
H : x < z
C : ~ x < y
______________________________________(1/1)
x < z
```

Now the conclusion is one of our hypotheses, so completing this subgoal is *trivial*. (literally).

```
trivial.
```

```
Proof completed.
```

That completes the proof. We should save it to be able to use it elsewhere.

```
Qed.
```

```
exLem is defined
```

Let's see how *autoSimpl* would have fared with this lemma.

```
Lemma exLem' : forall x y z: Z,
(if x <b y then y <b z else x <b z) = true -> x < z.
Proof.
intros.
autoSimpl.
```

```
Proof completed.
```

In this case, the lemma is easy enough for *autoSimpl* to complete.

```
Ltac trans H H0 T:= ...
```

This tactic is used to add new hypotheses by using transitivity on the first two arguments. The first two arguments need to be inequalities, say $H : A \leq B$ and $H_0 : B \leq C$ and T needs to be a hypothesis identifier (one that isn't used). It adds $T : A \leq C$ to Hyp. It also works if either of H or H_0 are strict inequalities.

```
Ltac define s def := ...
```

This tactic creates a new hypothesis defining s as def, i.e. we add $H : s = def$ to Hyp. Note that s must be an unused variable name.

```
Ltac remind t t':= ...
```

This tactic's name is short for remember induction. It creates a copy of t, then uses induction on t, calling the variable t'. When using induction by itself, the variable t disappears from the hypotheses. $remind$ lets us argue by cases and adds $t = case_i$ as a one of the hypotheses.

```
Ltac simplOr := ...
```

This tactic simplifies disjunctions that contain either true or false. E.g. simplifies $A \vee false$ to A and $A \vee true$ to $true$. Far after its creation, it was also been made to simplify conjunctions as well, but the name was unchanged.

```
Ltac leftb := ...
Ltac rightb := ...
```

These two tactics are the equivalent of $left$ and $right$ for disjunctions on bools. Recall that $left$ (resp. $right$) replaces a conclusion of type $A \vee B$ with A (resp. B).

```
Ltac matchInd H :=
match type of H with
| match ?x with | timeValue _ => _ end = true => induction x
| match ?x with | timeValue _ => _ end => induction x
...
end.

Ltac matchIndGoal :=
match goal with
| |- match ?x with | timeValue _ => _ end = _ => induction x
| |- _ = match ?x with | timeValue _ => _ end => induction x
...
end.
```

These two tactics automatically use induction on $srac$. They are used to prove that $sracCheck$ is symmetric.

5.2 Soundness

In this section, we will prove the soundness of the algorithm, that is if the algorithm finds a conflict ($conflict\_check$ is true, see page 45 for its definition), then there really is a conflict ($rule\_conflict$ is true, see page 38), i.e.

```
forall r1 r2: rule,
  conflict_check r1 r2 = true -> rule_conflict r1 r2.
```

Since rules have four *srac* fields, we will start by showing a similar lemma that only involve pairs of *srac*, that is, if the algorithm says there's an intersection between two *srac* ($sracCheck = true$), then there exists a request that matches both *srac*:

```
forall sr1 sr2: srac,  sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
```

We will do this by taking each pair of *srac* and proving soundness for that pair. If the algorithm is sound for each pair of specific *srac*, then it is sound. We start with the pair $timeInRange$ and $timeInRange$. Recall that $sracCheck$ for this pair uses $allCases$ to determine if there is an intersection between two ranges (see Figure 4.3 and Figure 4.4 for the definitions of $allCases$ and $sracCheck$). Also recall that $valueMatch$ was defined (in Figure 4.1) as $inRange\ req\ m\ M$ for time ranges (defined on page 36).

```
Definition inBothRanges:bool := (inRange req m1 M1) &&
                                (inRange req m2 M2).
```

Note that $inBothRanges$ is defined in a Coq section containing local variables m_1, M_1, m_2, M_2 and req, so they are not shown in the definition above, but will become arguments when used later (outside of its section). The implied definition outside of the section in shown below, however this version is not found in the code.

```
inBothRanges (m1 M1 m2 M2 req : Z) : bool :=
  inRange req m1 M1 && inRange req m2 M2
```

Unfolding all of the definitions, the soundness lemma for this pair is equivalent (assuming $m_1 \leq m_2$) to showing:

```
(allCases m1 M1 m2 M2 = true) ->
(exists req : Z, (inBothRanges m1 M1 m2 M2 req) = true).
```

for all arguments. We can do this by showing each case separately. Note that some parts of the code for the proof is omitted. The missing parts will be replaced by "...". We either remove these because they are just used to clean the proof (e.g. a clear command, to remove unneeded hypotheses, but otherwise doesn't affect the proof), or is too similar to another proof (or part of one) to need re-explaining. The entire proofs can be found in the appendices A-D.

```
Lemma case1Sound: case1 = true -> inBothRanges = true.
Proof.
decompose [and] constraints; clear constraints.
generalize rangeOfMax.
intros.
unfold case1, inBothRanges, inRange, inRange', MAX in *.
decompAndb H3.
decompAndb H5.
decompose [and] H; clear H.
repeat autoSimpl.
Qed.

Lemma case2Sound: case2 = true -> inBothRanges = true. ...
Lemma case3Sound: case3 = true -> inBothRanges = true. ...
Lemma case4Sound: case4 = true -> inBothRanges = true. ...
```

All four lemmas are proven in a similar way (using the same proof commands), thus we only show the proof of the first case. We start by adding in our time constraints (e.g. $0 \leq m_1 < MAX$), unfold all of the definitions and decompose hypotheses into simple ones (splitting any hypothesis of the form $A \wedge B$ into two hypotheses). All of the commands except the last one (repeat *autoSimpl*) are used for simplifying. Once we're down to simpler hypotheses, we use *autoSimpl* to complete the proof (see Section 5.1 for explanation on custom tactics). Even though *autoSimpl* completely finishes the proof automatically, we will show the steps, both to illustrate the power of this tactic as well as to provide insight into how the proof is done. For readability,

we will leave some definitions not unfold, while they are usually all unfolded at the beginning. Also, we will rearrange some hypotheses so they are in the group they came from. Also boolean inequalities (e.g. $(M2 <\text{b } MAX) = true$) will be automatically converted to their $Prop$ counterpart for readability (i.e. $b2PH$ will implicitly be called after every command). The following subgoal is the one just before $autoSimpl$ is applied (with some definitions folded).

```
H1 : 0 <= req          H2 : req < MAX
H0 : 0 <= m1           H4 : m1 < MAX
H7 : 0 <= m2           H9 : m2 < MAX
H11 : m1 <= M1         H12 : M1 < m1 + MAX
H5 : m2 <= M2          H13 : M2 < m2 + MAX

H3 : m1 <=b M2 - MAX  H8 : M2 - MAX <= M1
H6 : inRange req m1 M2 = true
______________________________________(1/1)
inRange req m1 M1 && inRange req m2 M2 = true
```

We have our time constraints at the top, followed by the unfolded $case1$ then our conclusion. For readability, we will omit hypotheses which have not been changed by the application of the previous command unless it will be used in the next step.

```
unfold inRange in *.
```

```
H6 : (if M2 <b MAX then inRange' req m1 M2
                   else inRange' req m1 (M2 - MAX)) = true
______________________________________(1/1)
(if M1 <b MAX then inRange' req m1 M1
              else inRange' req m1 (M1 - MAX)) &&
(if M2 <b MAX then inRange' req m2 M2
              else inRange' req m2 (M2 - MAX)) = true
```

Here we have some if statements. Our hypotheses don't let us conclude if they are true or false, so we will need to split the proof into cases. We start with the if statement in H_6. This condition defines what kind of range $m_2 - M_2'$ is. If the condition is true, the range is $[m_2, M_2']$, if it is false then we started with $M_2' < m_2$ and

had to convert the range to pass MAX. Hence the range will be $[0, M_2'] \bigcup [m_2, MAX)$. We start with the former.

```
caseSimpl (M2 <b MAX).

H0 : 0 <= m1    H3 : m1 <=b M2 - MAX

H6 : inRange' req m1 M2 = true
C : M2 < MAX
______________________________________(1/2)
(if M1 <b MAX then inRange' req m1 M1
              else inRange' req m1 (M1 - MAX)) &&
  inRange' req m2 M2 = true
```

That command simplified H_6 and also simplified the conclusion as it contained the same condition. There is a problem however, we have inconsistent hypotheses: $0 \leq m_1 \leq M_2 - MAX < 0$. We have a contradiction, and can conclude the subgoal. Now for the other subgoal (where the condition in H_6 is false). The condition being false means that the range loops past MAX.

```
H6 : inRange' req m1 (M2 - MAX) = true
C : ~M2 < MAX
______________________________________(1/1)
(if M1 <b MAX then inRange' req m1 M1
              else inRange' req m1 (M1 - MAX)) &&
  inRange' req m2 (M2 - MAX) = true
```

Here, we have another if statement in the conclusion. Again, our hypotheses don't let us conclude if it is true or false, so we split the proof into two more cases, this time on whether the first range is looping or not.

```
caseSimpl (M1 <b MAX).

C0 : M1 < MAX
______________________________________(1/2)
inRange' req m1 M1 && inRange' req m2 (M2 - MAX) = true
```

We start with the first range not looping past MAX. This means that we have to show $m_1 \leq req \leq M_1$ for req to be in the first range. The second range loops past MAX. Thus we need to show either $req \leq M_2 - MAX$ or $m_2 \leq req$ to show req is in that range. Unfolding *inRange*' and simplifying the conditions within them, we get that as a goal.

```
unfold inRange' in *.

H6: (if m1 <=b M2 - MAX
       then (m1 <=b req) && (req <=b M2 - MAX)
       else (req <=b M2 - MAX) || (m1 <=b req)) = true
______________________________________(1/2)
(if m1 <=b M1
  then (m1 <=b req) && (req <=b M1)
  else (req <=b M1) || (m1 <=b req)) &&
(if m2 <=b M2 - MAX
  then (m2 <=b req) && (req <=b M2 - MAX)
  else (req <=b M2 - MAX) || (m2 <=b req)) = true
```

We know that the first two *if* conditions are true (from our hypotheses), and that the third is false. We can use *test* to simplify them.

```
test (m1 <=b M2 - MAX).
test (m1 <=b M1).
test (m2 <=b M2 - MAX).

H1 : 0 <= req          H2 : req < MAX
H0 : 0 <= m1           H4 : m1 < MAX
H7 : 0 <= m2           H9 : m2 < MAX
H11 : m1 <= M1         H12 : M1 < m1 + MAX
H5 : m2 <= M2          H13 : M2 < m2 + MAX

H3 : m1 <= M2 - MAX    H8 : M2 - MAX <= M1
H6 : m1 <= req         H10 : req <= M2 - MAX

C : ~ M2 < MAX
C0 : M1 < MAX
______________________________________(1/2)
(m1 <= req) /\ (req <= M1) /\
  ((req <= M2 - MAX) \/ (m2 <= req))
```

From this point we can show that *req* is in both ranges: H_6, H_{10} and H_8 let us conclude *req* is in the first range, and H_{10} lets us conclude *req* is also in the second range, concluding this subgoal. This leaves us with our last case:

```
H1 : 0 <= req          H2 : req < MAX
H0 : 0 <= m1           H4 : m1 < MAX
H7 : 0 <= m2           H9 : m2 < MAX
H11 : m1 <= M1         H12 : M1 < m1 + MAX
H5 : m2 <= M2          H13 : M2 < m2 + MAX

H3 : m1 <= M2 - MAX    H8 : M2 - MAX <= M1
H6 : inRange' req m1 (M2 - MAX) = true

C0 : ~ M1 < MAX
C : ~ M2 < MAX
______________________________________(1/1)
inRange' req m1 (M1 - MAX) && inRange' req m2 (M2 - MAX) = true
```

We can unfold *inRange′* again and test the conditions contained within to arrive at:

```
H6 : m1 <= req         H14 : req <= M2 - MAX
______________________________________(1/1)
((req <= M1 - MAX) \/ (m1 <= req)) /\
  ((req <= M2 - MAX) \/ (m2 <= req))
```

For this last case, H_6 and H_{14} show that *req* is in both ranges, completing the proof. As stated, *autoSimpl* could have completed the proof after unfolding definitions (by following the steps we followed), however we wanted to show an example of how it works. Future applications of *autoSimpl* will use similar steps. First finding conditions it can simplify (such as in if statements), simplifying if it can, splitting up into cases if it can't. It repeats this until it is down to simple inequalities, in which case it can usually complete the proof.

```
Theorem timeRangeSound: (case1 || case2 || case3 || case4) = true
-> inBothRanges = true.
Proof.
intros.
```

```
b2PH;
[apply case1Sound |apply case2Sound
|apply case3Sound |apply case4Sound];
assumption.
Qed.
```

In this theorem, we put the four previous lemmas together. We do this by splitting it up into the four cases, then applying the corresponding lemma.

Again this lemma uses the variables m_1, M_1, m_2, M_2 and req implicitly, and these will become arguments to the lemma outside of its section.

```
Lemma rangeSound': forall m1 M1' m2 M2' M1 M2: Z,
0 <= m1 < MAX /\ 0 <= M1' < MAX /\ M1 = M'ToM m1 M1' /\
0 <= m2 < MAX /\ 0 <= M2' < MAX /\ M2 = M'ToM m2 M2' ->
(allCases m1 M1 m2 M2 = true) ->
(exists req : Z, (inBothRanges m1 M1 m2 M2 req) = true).
Proof.
...
[exists m1 | exists m2 | exists m2 | exists m1];
apply (timeRangeSound _ _ _ _ M1' M2' _);
...
Qed.
```

This lemma is similar to the previous lemma, however, it has the time constraints as an argument and it is expressed using $allCases$ instead of each separate cases. Recall that $allCases$ is defined using the alternate definition of the four cases ($case1'$, $case2'$, $case3'$ and $case4'$) which don't include the intersection range in their definition.

To prove this theorem, we will use the lemma $timeRangeSound$. In order to use that lemma, Coq needs a proof of $(case_1||case_2||case_3||case_4) = true$ as an argument. Since we assumed $allCases$ and thus $case'_i$ (the cases without the intersection range in definition), we can conclude $case_i$ (the cases with intersection) by choosing any req in the intersection range. We will use the lower bound of the range as this value.

We are now ready to prove soundness for this pair of $srac$ (2 time ranges). We need to prove the lemma found at the start of this section while restricting sr_1 and sr_2 to $timeInRange$.

```
Lemma rangeSound: forall (m1v M1'v m2v M2'v : tValue) (sr1 sr2: srac),
sr1 = timeInRange m1v M1'v -> sr2 = timeInRange m2v M2'v ->
sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
...
caseSimpl (m1 <=b m2);
...
[apply (rangeSound' _ M1' _ M2'); tauto | |
 apply (rangeSound' _ M2' _ M1'); tauto | ];
...
generalize (rangeEquiv x m1 M1 M1' H3 H2 HeqM1);
...
tauto.
Qed.
```

To restate what the lemma means, it shows that for this particular pair of *srac* (2 time ranges), if *sracCheck* says there's an intersection ($sracCheck = true$), then there is an intersection (there exists a request whose argument is in both ranges).

The proof proceeds by adding the time constraints, and unfolding the definitions. Since the conflict detection assumes that the first range has the smallest lower bound, we split the proof into two, depending on if $m_1 \leq m_2$ or not. Then for each case, we apply the previous theorem. We then have to apply *rangeEquiv* (which states that using M is equivalent to using M' in *inRange*) since *valueMatch* uses M' while *allCases* as well as *inBothRanges* uses M.

This concludes the soundness proofs for this specific pair. We will later use this lemma to prove that it works for any pair of *srac* instead of just this one pair. For now, we will move onto the next pair, *timeInRange* with *timeGe*. The proof for this pair is simpler than the previous pair, and can go straight to the proof of soundness for the pair.

```
Lemma rangeGeSound: forall (m1v M1'v m2v : tValue) (sr1 sr2: srac),
sr1 = timeInRange m1v M1'v -> sr2 = timeGe m2v ->
sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
```

```
Proof.
...
caseSimpl ((m1 <=b M1') && (M1' <b m2));
...
exists (timeReq (timeValue (MAX-1))).
...
exists (timeReq M1'v).
...
Qed.
```

Again, we use the lemma at the start of the section, restricting sr_1 and sr_2 (to *timeInRange* with *timeGe*). We start this proof by unfolding the definitions and adding our assumptions. Then we split up the proof into two parts (on page 41, we saw three cases, but the first two can be combined), there is an intersection either when $M_1' < m_1$ or if $m_2 \leq M_1'$. In the first case, anything greater or equal to both m_1 and m_2 is an intersection. Since they are both time values, $MAX - 1$ fits that criteria. In the second case, anything between m_2 and M_1 work, so we choose M_1.

```
Lemma rangeLtSound: forall (m1v M1'v M2v: tValue) (sr1 sr2: srac),
sr1 = timeInRange m1v M1'v -> sr2 = timeLt M2v ->
sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
...
caseSimpl ((M2 <=b m1) && (m1 <=b M1')); [exfalso; auto with * |].
...
exists (timeReq m1v);
...
exists (timeReq (timeValue (-1))).
...
Qed.
```

This is the proof for the next pair: *timeInRange* with *timeLt*. This lemma's proof is almost the same as the previous proof except the intersections and check for intersection changed. We update these and complete the proof in a similar way. Note again that we use -1 as an intersection. While it is not a real time value, it avoids having a special case for *timeLt* 0 which needs to be reported as an empty rule.

```
Lemma geSound: forall (m1v m2v: tValue) (sr1 sr2: srac),
sr1 = timeGe m1v -> sr2 = timeGe m2v -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
...
exists (timeReq (timeValue (MAX-1))).
omega.
Qed.
```

For this pair (*timeGe* with *timeGe*), $MAX - 1$ is always an intersection, and so we unfold the definitions, set $MAX - 1$ as an intersection and complete the proof.

```
Lemma geLtSound: forall (m1v M2v: tValue) (sr1 sr2: srac),
sr1 = timeGe m1v -> sr2 = timeLt M2v -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
...
exists (timeReq (timeValue m1)).
simpl. omega.
Qed.

Lemma ltSound: forall (M1v M2v: tValue) (sr1 sr2: srac),
sr1 = timeLt M1v -> sr2 = timeLt M2v -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
...
exists (timeReq (timeValue (-1))).
simpl. omega.
Qed.

Lemma intGtSound: forall (m1v m2v: iValue) (sr1 sr2: srac),
sr1 = intGt m1v -> sr2 = intGt m2v -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
...
caseSimpl (m1 <=b m2);
[exists (intReq(intValue (m2+1))) | exists (intReq(intValue (m1+1)))];
...
Qed.
```

```
Lemma anySound: forall (sr1 sr2: srac),
sr1 = any -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
...
Qed.
```

Above are the last few soundness lemmas for pairs of *srac*. They all have pretty simple proofs. *intGt* is the only one with something new. It needs a value greater than both m_1 and m_2. So we split the proof into two parts, one with m_1 larger and one with m_2 larger. Then we pick $max(m_1, m_2) + 1$ as an intersection. Also, *anySound* proves soundness for all five pairs which contain the *srac any*. It is a simple function that always applies to requests, thus always conflicts with other *srac*.

```
Hint Resolve rangeGeSound rangeSound geSound
 rangeLtSound geLtSound ltSound intGtSound anySound.
```

Now that we have soundness lemmas for each pair of type of *srac*, we add them to *Hint*. After this statement, the lemmas will be added to the lemmas tried by *eauto*, an auto-prover.

```
Definition rank : srac -> Z :=
fun sr =>
match sr with
| any => 0
| timeInRange _ _ => 1
| timeGe _ => 2
| timeLt _ => 3
| intGt _ => 4
end.
```

This assigns ranks (integers) to the each *srac*. The lemmas we proved only work if the *srac* are in the correct order (e.g. there is a lemma for the pair (timeGe, timeLt), but none for the pair (timeLt, timeGe)). This rank will tell the auto-provers which order to put each *srac* so the lemmas apply.

```
Lemma sracCheckSoundness: forall sr1 sr2: srac,
 sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
...
induction sr1;
...
induction sr2;
...
eauto.
Qed.
```

Now, we show that for any two *srac*, if we find a conflict, then there really is one. We use induction on both *srac*, giving us 25 subgoals, one for each pair of *srac*. If the two *srac* need different types, we can eliminate them (as they have no intersection). At this point, each pair matches one of our soundness lemmas. Because of the hints we added, *eauto* can complete the proof.

Since rules contain four lists of *srac* (see page 34), we have to prove that our conflict algorithm for lists (*listListCheck* found on page 45) is sound.

```
Lemma listCheckSoundness: forall l1 l2: list srac,
 listListCheck l1 l2 = true ->
exists reqv: reqValue, (listMatch reqv l1) /\ (listMatch reqv l2).
Proof.
...
induction l2.
exfalso; auto with *.
...
induction l1.
exfalso; auto with *.
...
generalize (sracCheckSoundness _ _ H);
...
generalize (IHl1 H);
...
generalize (IHl2 H);
...
Qed.
```

We proceed by induction first on l_2 then on l_1. Our base cases for induction are that the lists are empty, thus we complete the proof by contradiction (we assumed we found a conflict, which can't happen without any *srac*). Once past our base case, we can use the previous lemma followed by our inductive hypotheses to complete the proof.

```
Lemma conflict_check_soundness:
  forall r1 r2: rule,
    conflict_check r1 r2 = true -> rule_conflict r1 r2.
Proof.
...
generalize (listCheckSoundness _ _ H0); ...
generalize (listCheckSoundness _ _ H1); ...
generalize (listCheckSoundness _ _ H2); ...
generalize (listCheckSoundness _ _ H3); ...
induction a1; induction a2;
...
Qed.
```

Next, we test soundness when finding conflicts between two rules. Since rules are four lists of *srac* with an access action, we can simply use the previous lemma on each list. After that, we use induction on the access actions. In the cases that they are different, we have a conflict, as $listCheckSoundness$ showed the rules intersect.

Lastly, we prove soundness of the algorithm. The definition and theorem stated below are exactly the same as in [3]. Soundness of our algorithm is expressed with the same statement. We can use the same proof commands as was used in that paper, as we have the same properties about our rules and algorithm as they did (that is, $conflict\_check\_soundness$ is true for both). Because the reasoning is the same, we will simply state the soundness theorem and explain how the proof works. First, as in [3], we define what it means for the ith and jth rules in the set rs to be in conflict. This definition uses $rule\_conflict$, which is true if and only if its arguments (two rules) are in conflict (see page 38). Its arguments use the function Nth. $Nth\ rs\ i\ Hi$ returns the ith entry in the set rs but needs a proof (H_i) that i is a valid index in

rs. To ensure the indices are valid, it uses dependant conjunction which has notation $\{A**B\}$. Both the definition for Nth and of dependant conjunction can be found in [3] or a copy of the definitions in Appendix D.

```
Definition rs_conflict: rule_set -> nat -> nat -> Prop :=
fun rs i j =>
  {(i < (length rs))%nat ** fun Hi =>
  {(j < (length rs))%nat ** fun Hj =>
   (rule_conflict (Nth rs i Hi) (Nth rs j Hj))}}.
```

If $rs\_conflict\ rs\ i\ j$ is true then there is a conflict between the ith and jth rule in the rule set rs. Thus we need to prove that if our algorithm finds a conflict between the ith and jth rule (i.e. the pair (i, j) is contained in the list returned by $find\_conflicts$ found on page 48), that $rs\_conflict$ is true for that pair. I.e.

```
Theorem conflicts_soundness:
  forall (rs:rule_set)(i j:nat),
    In (i,j) (find_conflicts rs) -> rs_conflict rs i j.
```

We prove this theorem by using induction on the rule set, starting at the end, and then showing we can add rules to the front of the list and update indices (increase the old indices by one and add the new conflicts) to keep a valid list of conflict indices.

5.3 Completeness

In this section, we will show the completeness of the conflict algorithm. That is, if there is a conflict between two rules (*rule_conflict* is true, see page 38 for its definition), then the algorithm finds it (*conflict_check* is true, see page 45), i.e.

```
forall r1 r2: rule,
   rule_conflict r1 r2 -> (conflict_check r1 r2 = true).
```

As with the proofs for soundness, we will start by proving completeness for each pair of *srac*. That is for each pair of *srac* (*timeInRange*, *timeGe*, *timeLt* and *intGt*) we will need to show:

```
exists reqv:reqValue, (valueMatch reqv sr1) -> (valueMatch reqv sr2) ->
sracCheck sr1 sr2 = true.
```

We start this with the pair *timeInRange* with *timeInRange*. Recall that if *valueMatch* (found in Figure 4.1) is true then *inBothRanges* is true (see page 56). Also recall that *sracCheck* is true when *allCases* is true (see Figure 4.4) and that *allCases* is true based on the chart in Figure 4.2 and ranges found in Table 4.1.

We will start by showing that the 12 ranges in the table include every case. Note that the entry number (row, column) in the chart is commented out at the end of each line (Coq comments are between (* and *)).

```
Lemma permutations: m1 <= m2 ->
((( m1 <= M1' < m2 /\ m2 <= M2' ) \/ (*1,1*)
( m1 <= M1' <= M2' /\ M2' < m2 )) \/ (*2,1*)
(( m1 <= m2 <= M1' /\ M1' <= M2' ) \/ (*3,1*)
( m1 <= m2 <= M2' /\ M2' <= M1' )) \/ (*1,2*)
(( m1 <= M2' <= M1' /\ M1' < m2 ) \/ (*2,2*)
( m1 <= M2' < m2 /\ m2 <= M1' )))  (*3,2*)
\/
((( M1' < m1 <= m2 /\ m2 <= M2' ) \/ (*1,3*)
( M1' < m1 <= M2' /\ M2' < m2 )) \/ (*2,3*)
(( M1' <= M2' < m1 /\ m1 <= m2 ))) (*3,3*)
\/
```

```
((( M2' < m1 <= M1' /\ M1' < m2 ) \/ (*1,4*)
( M2' < m1 <= m2 /\ m2 <= M1' )) \/ (*2,4*)
( M2' <= M1' < m1 /\ m1 <= m2 )). (*3,4*)
Proof.
clear constraints.
intros.
omega.
Qed.
```

omega can solve this, however it takes a long time for it to do so. In the appendix, we show an alternate proof that splits the proof into four parts, each part having one of the values as the smallest, i.e. either m_1 is smallest or M_1' is smallest or m_2 is smallest or M_2' is smallest. Each of these four case falls into one of the groups, e.g. if m_1 is the smallest, then we fall into one of the first six permutations. If M_2' is the smallest, then we fall into the last three. Note that if m_2 is smallest then $m_1 = m_2$ (since we assume $m_1 \leq m_2$) and so it is similar to the case where m_1 is smallest.

In each of these four cases, we then choose the next smallest, splitting each case into three additional cases (e.g. we get three cases when m_1 is smallest: either M_1', m_2 or M_2' is smallest), at which time each case is trivial.

We are now ready to prove completeness on the unfolded definitions.

```
Theorem timeRangeComp:  m1 <= m2 -> (inBothRanges = true) ->
(case1 = true) \/ (case2 = true) \/ (case3 = true) \/ (case4 = true).
Proof.
...
generalize (permutations H); intro.
...
decompose [or] H8; clear H8;
...
[exfalso       | do 3 right        | do 2 right; left  |
right; left    | left              | hard1             |
right; left    | hard2             | do 2 right; left  |
elimtype False | do 2 right; left  | right; left       ];
...
Qed.
```

We start the proof by splitting up the proof into 12 cases, one per permutation, by generalizing the *permutations* lemma then decomposing the hypothesis which contains the permutations (H_8).

For each permutation, we choose the corresponding cases based on the chart in Figure 4.2. The 12 tactics in the square brackets correspond to choosing the corresponding case, e.g. $exfalso$ (Coq's command for contradiction) means there is no case for that permutation, $right; left$ corresponds to $case2$, $hard1$ and $hard2$ are for sections of the chart with more than one case. Once the right case has been chosen, auto-provers complete the proof.

Note that this theorem uses the variables m_1, M_1, m_2, M_2 and req implicitly, and they will become arguments outside of the section the theorem is defined.

As with the soundness lemmas, we have to convert this one to use $allCases$ (and so use $case'_i$ instead of $case_i$).

```
Lemma rangeComp' : forall m1 M1 m2 M2 M1' M2' req : Z,
inRange req m1 M1 = true -> inRange req m2 M2 = true -> m1 <= m2 ->
0 <= req < MAX /\
0 <= m1 < MAX /\ 0 <= M1' < MAX /\ M1 = M'ToM m1 M1' /\
0 <= m2 < MAX /\ 0 <= M2' < MAX /\ M2 = M'ToM m2 M2' ->
allCases m1 M1 m2 M2 = true.
Proof.
...
generalize (timeRangeComp m1 M1 m2 M2 M1' M2' req H2 H1 H3);
...
destruct H4.
do 3 leftb; b2P; trivial.
destruct H. do 2 leftb; rightb; b2P; trivial.
destruct H. leftb; rightb; b2P; trivial.
rightb; b2P; trivial.
Qed.
```

We start by using our previous theorem then match each corresponding case. Since $case'_i$ is contained in $case_i$, the proof is trivial after each case is matched.

We are now ready to state the final completeness lemma for this pair (two *srac* of type *timeInRange*). All that's left to prove is that we can convert from M to M' and that the previous lemma works when with $m_1 \leq m_2$ and when $m_2 < m_1$. We also need to state the lemma using *valueMatch* and *sracCheck* instead of the unfolded definitions.

```
Lemma rangeComp: forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(m1v M1'v m2v M2'v : tValue),
sr1 = timeInRange m1v M1'v -> sr2 = timeInRange m2v M2'v ->
 (valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.
Proof.
...
assert ( inRange req m1 M1 = inRange req m1 M1').
apply rangeEquiv; trivial.

assert ( inRange req m2 M2 = inRange req m2 M2').
apply rangeEquiv; trivial.
...
caseSimpl (m1 <=b m2); b2PH.
apply (rangeComp' _ _ _ _ M1' M2' req); tauto.
apply (rangeComp' _ _ _ _ M2' M1' req); try tauto.
omega.
Qed.
```

For this proof, we use our previous lemma on each case ($m_1 \leq m_2$ and $m_2 < m_1$) after having converted M to M' with *rangeEquiv* as we did with the soundness proof.

Next we prove completeness for the pair (*timeInRange*, *timeGe*).

```
Lemma rangeGeComp: forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(m1v M1'v m2v : tValue),
sr1 = timeInRange m1v M1'v -> sr2 = timeGe m2v ->
(valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.
Proof.
...
caseSimpl (m1 <=b M1'); [ | trivial].
caseSimpl (M1' <b m2); [ | trivial].
```

```
b2P.
exfalso.
omega.
Qed.
```

Recall that if we unfold *sracCheck* for this pair, then we get the following: `if (m1 <=b M1') && (M1' <b m2) then false else true`. This means that we can show that *sracCheck* is true by showing that $m_1 \leq M_1'$ and $M_1' < m_2$ can't both be true at the same time while having an intersection. We do this by cases on $m_1 \leq M_1'$ (if it is true or not). If it is false then the proof is trivial, as that directly implies that *sracCheck* is true (as wanted), leaving us with the true case. We do the same with $M_1' < m_2$ making that one true as well.

Unfolding *valueMatch*, we can conclude that $m_1 \leq req \leq M_1'$ from sr_1 and that $m_2 \leq req$ from sr_2. We now have a contradiction: $req \leq M_1' < m_2 \leq req$, which *omega* finds to complete the proof.

Next we have the pair (*timeInRange*, *timeLt*) which is nearly identical to the above proof.

```
Lemma rangeLtComp: forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(m1v M1'v M2v : tValue),
sr1 = timeInRange m1v M1'v -> sr2 = timeLt M2v ->
(valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.
Proof.
...
Qed.
```

And followed by the pair (*timeGe*, *timeGe*). Since *sracCheck* always unfolds to true for this pair, the proof is trivial. The same thing goes for (*timeLt*, *timeLt*), (*intGt*, *intGt*) and any pair containing *any*. Their statements are as follows:

```
Lemma geComp:  forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(m1v m2v : tValue),
sr1 = timeGe m1v -> sr2 = timeGe m2v ->
(valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
```

```
reqv = timeValue req -> sracCheck sr1 sr2 = true.

Lemma ltComp:  forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(M1v M2v : tValue),
sr1 = timeLt M1v -> sr2 = timeLt M2v ->
(valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.

Lemma intGtComp:  forall (reqv : iValue) (req : Z) (sr1 sr2: srac)
 (m1v m2v : iValue),
sr1 = intGt m1v -> sr2 = intGt m2v ->
(valueMatch (intReq reqv) sr1) -> (valueMatch (intReq reqv) sr2) ->
reqv = intValue req -> sracCheck sr1 sr2 = true.

Lemma anyComp: forall (reqv : reqValue) (sr1 sr2: srac),
sr1 = any -> sracCheck sr1 sr2 = true.
```

And finally, the pair (*timeGe*, *timeLt*).

```
Lemma geLtComp:  forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(m1v M2v : tValue),
sr1 = timeGe m1v -> sr2 = timeLt M2v ->
(valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.
Proof.
...
omega.
Qed.
```

This is another easy proof. After unfolding all the definitions, we find ourselves needing to prove $m_1 < M_2$ (from *sracCheck*) while knowing $m_1 \leq req$ and $req < M_2$ from the two *valueMatch* which omega does easily.

We are now ready to prove the completeness of the algorithm for any *srac*. We first add all the lemmas we just proved to *Hint*.

```
Hint Resolve rangeComp rangeGeComp geComp rangeLtComp
             geLtComp ltComp intGtComp anyComp.
```

```
Lemma sracCompleteness: forall (reqv : reqValue) (sr1 sr2: srac),
 (valueMatch reqv sr1) -> (valueMatch reqv sr2) ->
sracCheck sr1 sr2 = true.
Proof.
...
try (solve [diffTypesComp H Heqs]);
try (solve [diffTypesComp H0 Heqs0]);
swapIfSmallerComp Heqs Heqs0 s s0 C;
eauto.
Qed.
```

As we did with the soundness lemma, we use induction on sr_1 and sr_2 to get each pair as a separate case. We eliminate pairs with different types and swap sr_1 and sr_2 if needed to match our lemmas. After that *eauto* completes the rest of the pairs by automatically using our lemmas.

Now that we know the algorithm is complete for *srac*, we proceed like in soundness to lists of *srac* since XACML rules contain four lists of *srac* (see page 34).

```
Lemma listCompleteness: forall (reqv: reqValue) (l1 l2:list srac),
(listMatch reqv l1) -> (listMatch reqv l2) ->
(listListCheck l1 l2 = true).
Proof.
...
induction l2.
...
induction l1.
...
apply (sracCompleteness reqv); trivial.
rightb; apply IHl1; trivial.
rightb; apply IHl2; trivial.
Qed.
```

This proof is very similar to the proof for soundness. We again proceed by induction on both lists. The base cases are trivial as empty lists can't have any intersecting elements. We then apply our lemma and complete the proof with the induction hypotheses.

Lastly we prove completeness for rules.

```
Lemma conflict_check_completeness:
  forall r1 r2: rule,
     rule_conflict r1 r2 -> (conflict_check r1 r2 = true).
Proof.
...
generalize (listCompleteness _ _ _ H1 H2); intro;
generalize (listCompleteness _ _ _ H3 H7); intro;
generalize (listCompleteness _ _ _ H4 H8); intro;
generalize (listCompleteness _ _ _ H6 H10); intro;
rewrite H, H5, H0, H9, H11, H12; trivial.
Qed.
```

For this proof, all we need to do is unfold definitions and use our previous lemma for each of the four lists. Then use the fact that both $rule\_conflict$ and $conflict\_check$ need different access actions (permit and deny).

As with soundness, we take the remainder of the completeness proof directly from the firewall proof [3]. See pages 47 and 69 for definitions used in this theorem. In this case, we need to prove the following:

```
Theorem conflicts_completeness:
  forall (rs:rule_set)(i j:nat), (i<j)%nat ->
    rs_conflict rs i j -> In (i,j) (find_conflicts rs).
```

This theorem is just the converse of the soundness theorem with one exception: we force $i < j$. We do this for efficiency, as without it, every conflict would need to be found twice (as (i, j) then as (j, i)). The proof for completeness is similar to the one for soundness. We proceed by induction, showing we can update indices when needed.

Chapter 6

Expandability

This chapter will explain how to expand the code to add more functions and more types. This illustrates how to extend our work to the full XACML language. This includes adding new types, such as strings or double precision numbers and new XACML functions such as those using strings.

6.1 Adding new types

This section will explain how to add a new type. These new type won't do anything by themselves, but can be used later when making new functions.

To add a new type to the code, we need to add four things: an inductive type to define the new XACML type(similar to $iValue$ and $tValue$ found on page 32), a line in the definition of $reqValue$ (defined on page 33) so it can be recognized as a new type, and lines in two Ltacs ($matchInd$ and $matchIndGoal$ found on page 55) for the proofs to include the new type. Below, we will specifically see what we need to add in order to add a new type, say $nValue$, which takes an integer as an argument.

We must first add $nValue$ as a type with the following:

```
Inductive nValue: Set := newValue : Z -> nValue.
```

In this example, we only have one case, *newValue*, which takes an integer as an argument. We can add more cases as needed. *nValue* is the name of our new type, and *newValue* is the name of the constructor for this type. This line must be added before *value*.

Next we need to add a line in *reqValue*:

```
Inductive reqValue: Set :=
...
| newReq : nValue -> reqValue.
```

Again, *nValue* is the name of our new type, and *newVal*, is the name of the constructor for this value. The names for both constructors, as well as the type's name must be different.

Lastly, we add lines to the Ltacs *matchInd* and *matchIndGoal*:

```
Ltac matchInd H :=
...
| match ?x with | newValue _ => _ end = true => induction x
| match ?x with | newValue _ => _ end => induction x
end.

Ltac matchIndGoal :=
match goal with
...
| |- match ?x with | newValue _ => _ end = _ => induction x
| |- _ = match ?x with | newValue _ => _ end => induction x
end.
```

Here, we have to add the new type to be matched. We simply need to add these four lines for each one of our type's cases, needing only to change the constructor name (*newValue*) to the constructor for each case, followed by an _ for each argument in that constructor. In this example, we only have one case (*newValue*), thus we add four lines.

This ends adding the type. It doesn't do anything yet, since we haven't added any functions that use it, but the type is now ready to be used and all of the theorems still hold without modifying their proofs.

6.2 Adding a new *srac*

This section will explain how to add new *srac*.

To add a new *srac* (with existing types), one must add six things: one line in *srac* (defined on page 34) to define the constructors name and its arguments, a rank for the *srac* (defined on page 66), definitions expressing which requests apply to this *srac* in *valueMatch* (defined in Figure 4.1), checks for intersections between this *srac* and the existing *srac* in *sracCheck* (defined in Figure 4.4), theorems for soundness, completeness and symmetry of these checks, and hints to use these theorems. Below we will add a new function called *newFunc* using *nValue*, the type we just made, as the requests type. It will apply to any request whose value is greater than its argument's value, which will be an *iValue*. Unlike all of the other *srac* described previously, this *srac* will use more than one type.

First, we have to add the name and arguments in *srac*:

```
Inductive srac : Set :=
...
| newFunc : iValue -> srac.
```

In this example, we have one argument, but we could have had more than one, or even no arguments. As stated, it is called *newFunc* and has one argument of type *iValue*. The requests type will be defined later.

Next we add a line to *rank*:

```
Definition rank : srac -> Z :=
...
| newFunc _ => 1
end.
```

When adding ranks, we have to put an _ (each separated by spaces) for each argument that is needed by our *srac*. In this example, we have one argument, and so one _ following the *srac*'s name.

This rank defines which theorems need to be implemented. *srac* with a higher rank will need to implement theorems for soundness and completeness with all lower or equal ranked *srac* which need a request of the same type. In this case, it is a *srac* whose request type is a new type (later we will define it to be $nValue$), so we only need to write the theorems of soundness and completeness with itself. Had we made a function that needed integer requests, we would have needed to write theorems of completeness for $newFunc$ with $newFunc$ and ones for $newFunc$ with $intGt$.

Next we add the definition for which requests apply to our function $newFunc$ which use the type $newValue$ (which we just created in the previous section) in $valueMatch$:

```
Definition valueMatch: reqValue -> srac -> Prop := fun reqv sr =>
match reqv with
...
|  newReq (newValue req) =>
   match sr with
   | newFunc (intValue m) => Zgt req m
   | _ => False
   end
end.
```

As said above, our function applies to any request whose value, req, is greater than the functions argument, m. Zgt is a built-in function that tests if its first argument is greater than its second. `| _ => False` is used to eliminate functions that have the wrong request type. Should we be using a request of a type that's already used instead of $nValue$, we simply would have needed to add
`| newFunc (intValue m) => Zgt req m` to the corresponding block, instead of creating a new one.

Next, we add tests for intersections in sracCheck:

```
Definition sracCheck: srac -> srac -> bool := fun sr1 sr2 =>
match sr1 with
...
```

```
 | newFunc (intValue m1)  =>
   match sr2 with
   | newFunc (intValue m2) => true
   | _ => false
   end
end.
```

We need to add a conflict check for every pair of functions. Again we can dismiss all of the different types (which never have conflicts) with `| _ => false`. In this case, the only pair we need to worry about is the new function with itself. For this simple example, any arguments will have an intersection. This is because, any value that is bigger than both m_1 and m_2, the arguments, will apply to both *srac*, hence an intersection. Next we have to write theorems for soundness and completeness:

```
Lemma newSound: forall (m1 m2: iValue) (sr1 sr2: srac),
sr1 = newFunc m1 -> sr2 = newFunc m2 -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
...
Qed.

Lemma newComp:  forall (reqv: nValue)(m1 m2: iValue)(sr1 sr2: srac),
sr1 = newFunc m1 -> sr2 = newFunc m2 ->
(valueMatch (newReq reqv) sr1) -> (valueMatch (newReq reqv) sr2) ->
sracCheck sr1 sr2 = true.
Proof.
...
Qed.
```

These theorems will always stay the same, only differing in which functions $sr1$ and $sr2$ take. The request type and the number/type of arguments will also need to be changed, as they are bound to the functions.

The first lemma says that for a given pair of functions (both *newFuncs* in this example), if *sracCheck* says there's a conflict, then there truly is a conflict. The second says if there's a conflict, *sracCheck* finds it. When proving soundness with two different functions, $sr2$ needs to be the higher ranked function.

We also need theorems for symmetry:

```
Lemma newFuncSymmetric: forall (m1 m2: iValue)(sr1 sr2: srac),
sr1 = newFunc m1 -> sr2 = newFunc m2 ->
sracCheck sr1 sr2 = sracCheck sr2 sr1.
Proof.
...
Qed.
```

This says that if we change the order of $sr1$ and $sr2$, we get the same result. In some cases, this is trivially true, and will be proven automatically. In this case we can exclude this theorem completely, and the existing code will prove it automatically.

Lastly, we need to add hints so that the existing proofs use our new written theorems where they are needed. We do this using the following.

```
Hint Resolve newSound.
Hint Resolve newComp.
Hint Resolve newFuncSymmetric.
```

These can be added anywhere after the theorem has been proven and saved, as long as they appear before the lemmas that need them: *sracCheckSoundness*, *sracCompleteness* and *sracCheckSymmetric* (respectively). We can also add groups of hints with one command, by writing them all after one Hint Resolve command, separated by spaces. Because of this, it is recommended to add them in the same command as the existing hints (they are added in three places, one per lemma).

Chapter 7

Related Work

As opposed to many other access control languages, not much work has been done in XACML with respect to conflict detection. Ours is the first formally verified algorithm for finding conflicts.

Huonder [10] has done some work in finding conflicts in XACML, but had a different approach. In the work, he constructed generic conflict algorithms, which means that it can detect any conflict, however they need to be extended with definitions (of XACML functions and intersections) in order to work on concrete examples. He also proposed two ways to resolve the conflicts. The first was similar to what XACML already does. The second was to replace the current rule set with an equivalent rule set, in which every request that was previously permitted (respectively denied) is still permitted (respectively denied) but that rules do not intersect. This leads to a rule set that is free of conflicts, as each request can match at most one rule. In contrast, our approach provides an opportunity for the administrator to examine each conflict separately which is why we simply report the conflicts and do not try to modify the rule set.

In [14] a method for intersecting policy files is proposed. This article is targeted towards two companies trying to merge their policies, in order to get a combined

policy which includes the rules from both companies.

In [8], the authors do a change-impact analysis in which they detect the set of requests which have their access action changed when the rule set is modified. They do this by converting both versions of the XACML policies into a form of decision-diagram and do an analysis on these diagrams. For example, if we were to add a rule to the rule set, the algorithm would do an analysis and report the set of requests that have had their access action changed from permit to deny (or from deny to permit). While this algorithm wasn't designed to find conflicts, it could be extended to do so by removing rules one at a time. Each time a change is detected, the rule removed is in conflict with at least one other rule in the set. The intersection between the removed rule and the conflicting rules is the set of requests the algorithm reported as being changed. A drawback to doing this is that you don't know which rules the removed rule conflicts with. Another drawback is that this algorithm can't be used to find redundant rules. In our algorithm, we can find the intersection between the rules regardless of whether they are conflicting or redundant.

If we consider policy specification languages other than XACML, much work has been done for analyzing policies. We mention the most related here, which directly deal with conflict detection. One example is the firewall conflict detection algorithm on which this thesis is based [3]. In this paper, the authors describe a conflict detection algorithm between CISCO firewall rules. They then formally prove its correctness. Our work generalizes this work. CISCO ACL is a much simpler specification language than XACML. It is only targeted towards one application, as opposed to XACML which is standardized and is applicable in many more situations.

In [5], the author proposes a generic conflict algorithm which can be modified to work with any policy language in which policies are formulated as sets of rules. This algorithm needs to be extended with definitions in order to be used with specific languages. Our work goes much deeper into XACML than this paper does, allowing our algorithm to be used in actual policies (although simple ones at the moment).

Chapter 8

Conclusion and Future Work

In this thesis, we proposed an algorithm to find conflicts between rules defined in XACML. We then proved that this algorithm is both sound and complete.

Future work There are several ways to improve our work.

The first is to continue expanding on the algorithm adding more types and more functions, allowing the algorithm to work in more cases. We have only taken a small subset of the functions and types that XACML allows, thus expanding in this direction would make the algorithm more useful.

XACML is only one of many existing policy specification languages. Others include SecPAL [2], a language that uses boolean formulas to specify its access rules. In order for a request to be permitted, the request instantiates the variables in the rules, which then evaluates to true if and only if the request is permitted. Another language is LoPSiL [7], which is a location based specification language. A LoPSiL user will have software on a mobile device, and the rules change based on location input from the device. Ponder [6] is another language which is declarative, strongly-typed and object oriented, which makes it useful in many situations. Because there are many different access control policy specifying languages, future work includes generalizing our algorithm so that it can work with many more languages, and not

just XACML. As many of the concepts are closely related to XACML, but still very general (such as *srac*, which one can augment to use any function that requires an argument in the form of a request, not just functions found in XACML) this wouldn't be too difficult. All that needs to be implemented are definitions of rules and how they use *srac*, then prove correctness properties about these new rules.

Another direction we can take is to test our algorithm out on real life policies. As it doesn't work with all XACML functions at the moment, this would have to be after extending the algorithm.

While the Java program is usable, it has not been verified. In [3], they extracted their conflict detection algorithm to OCaml. Future work includes extracting the verified algorithm to be assured of its correctness. The Java version's parser has been modified to create a Coq file which can be loaded directly into Coq where the verified algorithm can be used to find conflicts, however this needs Coq installed in order to be used. It is one step towards automating the algorithm.

One more direction is to update the Coq code to return the intersection information, instead of just stating that two rules intersect. This would also allow a verified version of inter-policy conflict detection.

Appendix A

Tactics.v

This appendix contains the code for the stand alone tactics as well as the lemmas needed to use them.

```
Require Export ZArith.
Require Export Bool.

Open Scope Z_scope.

Infix "<=b" := Zle_bool (at level 70): Z_scope.
Infix "<b" := Zlt_bool (at level 70): Z_scope.

(* a few simple lemmas used by tactics *)
Lemma flip': forall A B:Z, 0 <= A -> ~A + B < B.
Proof.
intros; omega.
Qed.

Lemma flip: forall A B:Z, 0 <= A -> (A + B <b B) = false.
Proof.
intros.
generalize (flip' A B H); intro.
generalize (Zlt_is_lt_bool (A + B) B); intro.
case_eq (A + B <b B); intro;
tauto.
Qed.

Lemma pmB: forall A B:Z, A + B - B = A.
```

```
Proof.
intros; omega.
Qed.

Lemma Zlt_bool_imp_lt : forall n m:Z, Zlt_bool n m = true -> (n < m).
Proof.
intros.
generalize (Zlt_is_lt_bool n m).
intuition.
Qed.

Lemma Zlt_imp_lt_bool : forall n m:Z, (n < m) -> Zlt_bool n m = true.
Proof.
intros.
generalize (Zlt_is_lt_bool n m).
intuition.
Qed.

Lemma andbSplit :
forall b1 b2:bool, (b1 && b2) = true -> b1 = true /\ b2 = true.
Proof.
auto with *.
Qed.

Lemma Nle_Nlebool: forall A B: Z, ~A <= B -> (A <=b B = false).
Proof.
intros.
case_eq (A <=b B).
intros.
generalize (Zle_cases A B); intro.
rewrite H0 in H1.
tauto.
trivial.
Qed.

Lemma Nlt_Nltbool: forall A B: Z, ~A < B -> (A <b B = false).
Proof.
intros.
case_eq (A <b B).
intros.
generalize (Zlt_cases A B); intro.
rewrite H0 in H1.
tauto.
trivial.
```

```
Qed.

Lemma eqLe : forall A B:Z, (A = B) -> (A <=b B) = true.
intros.
generalize (Zle_imp_le_bool A B).
auto with *.
Qed.

Lemma splitB : forall b1 b2 : bool,
b1 = true -> b2 = true -> (b1 && b2) = true.
Proof.
auto with *.
Qed.

Lemma orOrb: forall  b1 b2 : bool,
b1 = true \/ b2 = true -> (b1 || b2) = true.
Proof.
auto with *.
Qed.

Lemma LeNe: forall A B:Z, A <= B -> ~A = B -> A < B.
Proof.
intros.
omega.
Qed.

Lemma LeNe2: forall A B:Z, B <= A -> ~A = B -> B < A.
Proof.
intros.
omega.
Qed.

Ltac caseSimpl b :=
case_eq b;
let C := fresh "C" in
intro C;
try rewrite C in *.

Lemma NLtB_NLt: forall A B: Z, (A <b B) = false -> ~A < B.
Proof.
intros.
generalize (Zlt_is_lt_bool A B); intro.
caseSimpl (A <b B).
auto with *.
```

```
intuition.
Qed.

Lemma NLeB_NLe: forall A B: Z, (A <=b B) = false -> ~A <= B.
Proof.
intros.
generalize (Zle_is_le_bool A B); intro.
caseSimpl (A <=b B).
auto with *.
intuition.
Qed.

Lemma orTrue: forall b:bool, (b || true) = true.
Proof.
intros.
auto with *.
Qed.

Lemma andTrue: forall b:bool, (b && true) = b.
Proof.
intros.
induction b; trivial.
Qed.

Lemma orFalse: forall b:bool, (b || false) = b.
Proof.
intros.
induction b; trivial.
Qed.

Lemma andFalse: forall b:bool, (b && false) = false.
Proof.
intros.
induction b; trivial.
Qed.

Lemma deMorganOr: forall (a b: bool),
(a || b) = false -> a = false /\ b = false.
Proof.
induction a; induction b; tauto.
Qed.

Lemma deMorganAnd: forall (a b: bool),
(a && b) = false -> a = false \/ b = false.
```

```
Proof.
induction a; induction b; tauto.
Qed.

Ltac rewriteSimpl H :=
rewrite H in *;
match type of H with
| ( _ =  ?A + ?B) => generalize (pmB A B); intro temp;
                     rewrite temp in *; clear temp
| _ => idtac
end.

Ltac decompAndb H :=
generalize (andbSplit _ _ H); intro temp; destruct temp; clear H.

Ltac test H :=
match H with
  | ?A + ?B <b _ =>
    match goal with
    | x : 0 <= A |- _  => generalize (flip A B x); intro temp;
                          try rewrite temp in *; clear temp
    end
  | ?A <=b ?A => generalize (Zle_refl A) ; intro temp;
                 generalize (Zle_imp_le_bool _ _ temp); intro temp2;
                 try rewrite temp2 in *; clear temp temp2
  | ?A <=b ?B =>
    match goal with
    | x : ~A <= B |- _ => generalize (Nle_Nlebool _ _ x); intro temp;
                          try rewrite temp in *; clear temp
    | x : B <   A |- _ => generalize (Zlt_not_le _ _ x) ; intro temp;
                          generalize (Nle_Nlebool _ _ temp); intro temp2;
                          try rewrite temp2 in *; clear temp temp2
    | x : A <= B |- _ => generalize (Zle_imp_le_bool _ _ x); intro temp;
                         try rewrite temp in *; clear temp
    | x : A <   B |- _ => generalize (Zlt_le_weak _ _ x) ; intro temp;
                          generalize (Zle_imp_le_bool _ _ temp);
                          intro temp2; try rewrite temp2 in *;
                          clear temp temp2
    | x : A =   B |- _ => generalize (eqLe _ _ x) ; intro temp;
                          try rewrite temp in *; clear temp
    end

  |  ?A <b ?B =>
    match goal with
```

```
    | x : ~A < B |- _ => generalize (Nlt_Nltbool _ _ x); intro temp;
                         try rewrite temp in *; clear temp
    | x : B < A |- _ => generalize (Zlt_le_weak _ _ x) ; intro temp;
                        generalize (Zle_not_lt _ _ temp) ; intro temp2;
                        generalize (Nlt_Nltbool _ _ temp2); intro temp3;
                        try rewrite temp3 in *; clear temp temp2 temp3
    | x : B <= A |- _ => generalize (Zle_not_lt _ _ x) ; intro temp;
                         generalize (Nlt_Nltbool _ _ temp); intro temp2;
                         try rewrite temp2 in *; clear temp temp2
    | x : A <   B |- _  => generalize (Zlt_imp_lt_bool _ _ x); intro temp;
                           try rewrite temp in *; clear temp
    end
  | ?A = ?B => first[test A| test B]
  | if ?A then _ else _ => test A
  | ?A = ?B => first[test A| test B]
  | ?A && ?B => first[test A| test B]
  | ?A \/ ?B => first[test A| test B]

  | ?A && ?B => first[test A| test B]
  | ?A /\ ?B => first[test A| test B]
  | ?A || ?B => first[test A| test B]
  | ?A \/ ?B => first[test A| test B]
end.

Ltac testAll :=
match goal with
  | x : ?A = ?B |- _ => first[test A| test B]; try rewriteSimpl x
  | |- ?A = ?B => first[test A| test B]
  | [ x : if ?a then _ else _ |- _] => test a; try rewriteSimpl x
  | [ |- ?A \/ ?B] => first[test A| test B]
end.

Ltac b2PH :=
repeat
match goal with
  | [ x : (?a && ?b) = true |- _ ] => generalize (andbSplit _ _ x);
                                      clear x; intro x; destruct x
  | [ x : (?a || ?b) = true |- _ ] => generalize (orb_prop _ _ x);
                                      clear x; intro x; destruct x
  | [ x : (?a && ?b) = false |- _ ] => generalize (deMorganAnd _ _ x);
                                       clear x; intro x; destruct x
  | [ x : (?a || ?b) = false |- _ ] => generalize (deMorganOr _ _ x);
                                       clear x; intro x; destruct x
  | [ x : (?a <b ?b) = true |- _ ] => generalize (Zlt_bool_imp_lt _ _ x);
```

```
                                        clear x; intro x
  | [ x : (?a <=b ?b) = true |- _ ] => generalize (Zle_bool_imp_le _ _ x);
                                        clear x; intro x

  | [ x : (?a <b ?b) = false |- _ ] => generalize (NLtB_NLt _ _ x);
                                        clear x; intro x
  | [ x : (?a <=b ?b) = false |- _ ] => generalize (NLeB_NLe _ _ x);
                                         clear x; intro x
end.

Ltac b2P :=
b2PH;
repeat
match goal with
  | [ |- ((?a && ?b) = true)] => apply (splitB a b)
  | [ |- ((?a || ?b) = true)] => apply (orOrb a b)
  | [ |- ((?a <b ?b) = true)] => apply (Zlt_imp_lt_bool a b)
  | [ |- ((?a <=b ?b) = true)] => apply (Zle_imp_le_bool a b)
end.

Ltac simplOrIf :=
match goal with
  | [ x : _ = (if ?a then _ else _) |- _] => caseSimpl a;
                                            try rewriteSimpl x
  | [ _ : (if ?a then _ else _) = true |- _] => caseSimpl a
  | [ |- (if ?a then _ else _) = true ] => caseSimpl a
  | [ |- ?a = true \/ _ ] => caseSimpl a; [left; trivial| right]
end.

Ltac autoSimpl :=
try b2PH;
repeat testAll;
try simplOrIf;
try b2P;
trivial;
try tauto;
try omega.

Ltac trans H H0 T:=
   match type of H with
   | _ <= _ =>
      match type of H0 with
      |  _ <= _  => generalize (Zle_trans _  _ _ H H0) as T
      |  _ <   _  => generalize (Zle_lt_trans _  _ _ H H0) as T
```

```
      end
   | _ < _  =>
      match type of H0 with
      | _ <= _ => generalize (Zlt_le_trans _  _ _ H H0) as T
      | _ <   _  => generalize (Zlt_trans _  _ _ H H0) as T
      end
   end; intro.

Ltac define s def :=
set (temp := def);
remember temp as s;
unfold temp in *; clear temp.

Ltac remind t t':=
remember t as t';
induction t as [temp];
match goal with
| x : t' = _ |- _ => rename t' into t; rename temp into t'; rewrite x in *
end.

Lemma posGtNeg: forall (A B MAX: Z),
0 <= A -> B < MAX -> ((A <=b B - MAX) = false).
Proof.
intros.
caseSimpl (A <=b B - MAX).
b2PH.
exfalso.
omega.
trivial.
Qed.

Lemma maxGtSmall: forall (A B MAX: Z),
A < MAX -> 0 <= B -> ((A <=b B + MAX) = true).
Proof.
intros.
caseSimpl (A <=b B + MAX).
trivial.
b2PH.
exfalso.
omega.
Qed.

Ltac simplOr' H :=
match H with
```

```
| ?A || true => generalize (orTrue A); intro temp; rewrite temp; clear temp
| ?A || false => generalize (orFalse A); intro temp; rewrite temp;
                 clear temp
| ?A || ?B => simplOr' A; simplOr' B
| ?A && true => generalize (andTrue A); intro temp; rewrite temp;
                clear temp
| ?A && false => generalize (andFalse A); intro temp; rewrite temp;
                 clear temp
| ?A && ?B => simplOr' A; simplOr' B
| ?A <=b ?B - ?MAX => assert ((A <=b B - MAX) = false) as temp;
                      [apply (posGtNeg A B MAX); trivial |
                       rewrite temp; clear temp]
| ?A <=b ?B + ?MAX =>
  match A with  _ + _ => idtac | _-_ => idtac
  | _ => assert ((A <=b B + MAX) = true) as temp;
         [apply (maxGtSmall A B MAX); trivial | rewrite temp; clear temp]
  end
| _ => idtac
end.

Ltac simplOr :=
simpl;
match goal with
| |- ?A = ?B => simplOr' A; simplOr' B
end;
simpl.

Lemma leSwap: forall A B C :Z, (A + C <=b B) = (A <=b B - C).
Proof.
intros.
generalize (Zle_plus_swap A B C); intros.
caseSimpl (A + C <=b B);
caseSimpl (A <=b B - C);
b2PH;
tauto.
Qed.

Lemma leftBool: forall b1 b2: bool, (b1 = true) -> ((b1 || b2) = true).
Proof.
auto with *.
Qed.

Lemma rightBool: forall b1 b2: bool, (b2 = true) -> ((b1 || b2) = true).
Proof.
```

```
auto with *.
Qed.

Ltac leftb := apply leftBool.
Ltac rightb := apply rightBool.
```

Appendix B

TimeRange.v

This appendix, will contain the code for implementing and proving correctness of the *timeInRange srac*. At this point, it only contains the methods itself and has nothing to do with XACML. All parts relating to XACML and the other *srac* are found in the next appendix.

```
Require Export Tactics.

Open Scope Z_scope.
Section TimeInRange.

Definition MAX := 86400.

Definition inRange': Z -> Z -> Z -> bool :=
fun req m M' =>
  if (m <=b M') then (m <=b req) && (req <=b M')
                        else  (req <=b M') || (m <=b req).

Definition inRange: Z -> Z -> Z -> bool :=
fun req m M =>
 if (M <b MAX) then inRange' req m M
                          else inRange' req m (M-MAX).

Definition M'ToM : Z -> Z -> Z :=
fun m M' => if (m <=b M') then M' else M' + MAX.

Theorem rangeEquiv:  forall req m M M':Z,
```

```
0 <= m < MAX -> 0 <= M' < MAX -> M = M'ToM m M' ->
(inRange req m M = inRange req m M').
Proof.
unfold inRange; unfold inRange'.
intros.
destruct H.
destruct H0.
unfold M'ToM in H1.
caseSimpl(m <=b M').
rewriteSimpl H1.

generalize (Zlt_imp_lt_bool _ _ H3); intros; rewrite H4 in *.
rewrite C in *.
tauto.

generalize (flip _ MAX H0); intro.
rewriteSimpl H1.
rewrite H4 in *.
generalize (Zlt_imp_lt_bool _ _ H3); intros; rewrite H5 in *.
rewrite C in *.
tauto.
Qed.

Variables m1 M1 m2 M2 M1' M2' req:Z.

Definition case1 : bool := (m1 <=b M2 - MAX) && (M2 - MAX <=b M1) &&
                           (inRange req m1 M2).
Definition case2 : bool := (m1 <=b M2) && (M2 <=b M1) && (m1 <=b m2) &&
                           (inRange req m2 M2).
Definition case3 : bool := (m1 <=b m2) && (m2 <=b M1) && (M1 <=b M2) &&
                           (inRange req m2 M1).
Definition case4 : bool := (M1 <=b M2 - MAX) && (inRange req m1 M1).

Definition inBothRanges:bool := (inRange req m1 M1) && (inRange req m2 M2).

Hypothesis constraints:
0 <= req < MAX /\
0 <= m1   < MAX /\ 0 <= M1' < MAX /\ M1 = M'ToM m1 M1' /\
0 <= m2   < MAX /\ 0 <= M2' < MAX /\ M2 = M'ToM m2 M2'.

Lemma rangeOfMax: m1 <= M1 < m1 + MAX /\
                  m2 <= M2 < m2 + MAX.
Proof.
decompose [and] constraints; clear constraints.
```

```
split;
unfold M'ToM in *;
[caseSimpl (m1 <=b M1') | caseSimpl (m2 <=b M2')];
b2PH;
omega.
Qed.

(* Soundness *)
Lemma case1Sound: case1 = true -> inBothRanges = true.
Proof.
decompose [and] constraints; clear constraints.
clear dependent M1'.
clear dependent M2'.
generalize rangeOfMax.
intros.
unfold case1, inBothRanges, inRange, inRange', MAX in *.
decompAndb H3.
decompAndb H5.
decompose [and] H; clear H.
repeat autoSimpl.
Qed.

Lemma case2Sound: case2 = true -> inBothRanges = true.
Proof.
decompose [and] constraints; clear constraints.
clear dependent M1'.
clear dependent M2'.
generalize rangeOfMax.
intros.
unfold case2, inBothRanges, inRange, inRange', MAX in *.
decompAndb H3.
decompAndb H5.
decompose [and] H; clear H.
repeat autoSimpl.
Qed.

Lemma case3Sound: case3 = true -> inBothRanges = true.
Proof.
decompose [and] constraints; clear constraints.
clear dependent M1'.
clear dependent M2'.
generalize rangeOfMax.
intros.
```

```
unfold case3, inBothRanges, inRange, inRange', MAX in *.
decompAndb H3.
decompAndb H5.
decompose [and] H; clear H.
repeat autoSimpl.
Qed.

Lemma case4Sound: case4 = true -> inBothRanges = true.
Proof.
decompose [and] constraints; clear constraints.
clear dependent M1'.
clear dependent M2'.
generalize rangeOfMax.
intros.
unfold case4, inBothRanges, inRange, inRange', MAX in *.
decompAndb H3.
decompose [and] H; clear H.
repeat autoSimpl.
Qed.

Theorem timeRangeSound: (case1 || case2 || case3 || case4) = true
-> inBothRanges = true.
Proof.
intros.
b2PH;
[apply case1Sound
|apply case2Sound
|apply case3Sound
|apply case4Sound];
assumption.
Qed.

(* case 1 and 3 can happen at the same time, this proves they get
   2 distinct intersection ranges *)
Theorem case1and3distinct: (case1 && case3) = true -> False.
Proof.
decompose [and] constraints; clear constraints.
clear dependent M1'.
clear dependent M2'.
generalize rangeOfMax.
unfold case1, case3, inBothRanges, inRange, inRange', MAX in *.

intros.
decompose [and] H; clear H.
```

```
repeat autoSimpl.
Qed.

(* Completeness *)
Ltac hard1 :=
generalize (dec_Zle req M2'); intro temp;
destruct temp; [left | do 2 right; left].

Ltac hard2 :=
generalize (dec_Zle req M1'); intro temp;
destruct temp; [do 2 right; left|
generalize (dec_Zle req M2'); intro temp;
destruct temp; [left |do 2 right; left]].

Lemma smallest3: forall A B C:Z,
( A <= B /\ A <= C)\/
( B <= A /\ B <= C)\/
( C <= A /\ C <= B ).
Proof.
intros.
clear constraints.
omega.
Qed.

Lemma smallest4: forall A B C D:Z,
( A <= B /\ A <= C /\ A <= D )\/
( B < A /\ B <= C /\ B < D )\/
( C <= A /\ C <= B /\ C <= D )\/
( D < A /\ D <= B /\ D < C ).
Proof.
intros.
clear constraints.
omega.
Qed.

Lemma permutations: m1 <= m2 ->
((( m1 <= M1' < m2 /\ m2 <= M2' ) \/ (*1,1*)
( m1 <= M1' <= M2' /\ M2' < m2 )) \/ (*2,1*)
(( m1 <= m2 <= M1' /\ M1' <= M2' ) \/ (*3,1*)
( m1 <= m2 <= M2' /\ M2' <= M1' )) \/ (*1,2*)
(( m1 <= M2' <= M1' /\ M1' < m2 ) \/ (*2,2*)
( m1 <= M2' < m2 /\ m2 <= M1' )))  (*3,2*)
\/
((( M1' < m1 <= m2 /\ m2 <= M2' ) \/ (*1,3*)
```

```
( M1' < m1 <= M2' /\ M2' < m2 )) \/ (*2,3*)
(( M1' <= M2' < m1 /\ m1 <= m2 ))) (*3,3*)
\/
((( M2' < m1 <= M1' /\ M1' < m2 ) \/ (*1,4*)
( M2' < m1 <= m2 /\ m2 <= M1' )) \/ (*2,4*)
( M2' <= M1' < m1 /\ m1 <= m2 )). (*3,4*)
Proof.
clear constraints.
intros.

generalize (smallest4 m1 M1' m2 M2'); intros.
decompose [or] H0; clear H0; [left | right; left | left | do 2 right].

generalize (smallest3 M1' m2 M2'); intros.
decompose [or] H0; clear H0;
[destruct H2; generalize (Zle_lt_or_eq _ _ H0); intros; destruct H3;
  [left | right; left]
| right;left
| destruct H3; generalize (Zle_lt_or_eq _ _ H2); intros; destruct H3;
  [do 2 right | right; left] ];
omega.

generalize (smallest3 m1 m2 M2'); intros.
decompose [or] H0; clear H0;
[left | left | ]; omega.

right; left; omega.
omega.
Qed.

Theorem timeRangeComp:  m1 <= m2 -> (inBothRanges = true) ->
(case1= true) \/ (case2= true) \/ (case3= true) \/ (case4= true).
Proof.
intros.
unfold inBothRanges in H0.
b2PH.
decompose [and] constraints; clear constraints.
generalize rangeOfMax; intro.
decompose [and] H8; clear H8.
unfold inRange, inRange', MAX in *.

generalize (permutations m1 M1' m2 M2' H); intro.
unfold M'ToM, MAX in *.
```

```
decompose [or] H8; clear H8;

try rename H16 into H20;
decompose [and] H20; clear H20;
rename H21 into T12; rename H22 into T23; rename H16 into T34;
trans T12 T23 T13; trans T23 T34 T24; trans T13 T34 T14;

[exfalso       | do 3 right        | do 2 right; left  |
right; left    | left              | hard1             |
right; left    | hard2             | do 2 right; left  |
elimtype False | do 2 right; left  | right; left       ];

unfold case1, case2, case3, case4, inRange, inRange', MAX;
repeat autoSimpl;
exfalso;
auto with *.
Qed.

End TimeInRange.
```

Appendix C

ConflictDetect.v

This appendix contains the code for the implementations of XACML in Coq, the conflict detection algorithm between XACML rules and the proof of correctness for that algorithm.

```
Require Export TimeRange.

Require Export Arith.
Require Export List.
Open Scope Z_scope.

Infix "<=b" := Zle_bool (at level 70): Z_scope.
Infix "<b" := Zlt_bool (at level 70): Z_scope.
Section conflict.

Inductive access: Set := permit | deny.

Inductive tValue: Set := timeValue : Z -> tValue.
Inductive iValue: Set := intValue : Z -> iValue.

Hypothesis timeConst: forall tv : tValue,
match tv with
  | timeValue t => 0 <= t < MAX
end.

Inductive reqValue: Set :=
| timeReq : tValue -> reqValue
| intReq : iValue -> reqValue.
```

```
Inductive srac : Set :=
| any : srac
| timeInRange : tValue -> tValue -> srac
| timeGe : tValue -> srac
| timeLt : tValue -> srac
| intGt : iValue -> srac.

Definition rank : srac -> Z :=
fun sr =>
match sr with
| any => 0
| timeInRange _ _ => 1
| timeGe _ => 2
| timeLt _ => 3
| intGt _ => 4
end.

Record rule: Set := ruleCons {
  acc:access;
  subjects: list srac;
  resources: list srac;
  actions: list srac;
  conditions: list srac
}.

Record request: Set := requestCons {
 subj : reqValue;
 res: reqValue;
 acti: reqValue;
 cond: reqValue
}.

Definition valueMatch: reqValue -> srac -> Prop :=
fun reqv sr =>
match reqv with
|  timeReq (timeValue req) =>
   match sr with
   | any => True
   | timeInRange (timeValue m) (timeValue M) => inRange req m M = true
   | timeGe (timeValue m) => Zge req m
   | timeLt (timeValue M) => Zlt req M
   | _ => False
   end
```

```
|  intReq (intValue req) =>
   match sr with
   | any => True
   | intGt (intValue m) => Zgt req m
   | _ => False
   end
end.

Fixpoint listMatch  (reqv: reqValue) (ls:list srac) {struct ls}  : Prop :=
  match ls with
  | nil => False
  | cons sr ls' => (valueMatch reqv sr) \/ (listMatch reqv ls')
  end.

Hypothesis hasOneMatch: forall sr : srac,
  exists reqv: reqValue, valueMatch reqv sr.

Definition rule_permit: request -> rule -> Prop :=
fun rq rl =>
  acc rl = permit /\
  listMatch (subj rq) (subjects rl) /\
  listMatch (res rq) (resources rl) /\
  listMatch (acti rq) (actions rl) /\
  listMatch (cond rq) (conditions rl).

Definition rule_deny: request -> rule -> Prop :=
fun rq rl =>
  acc rl = deny /\
  listMatch (subj rq) (subjects rl) /\
  listMatch (res rq) (resources rl) /\
  listMatch (acti rq) (actions rl) /\
  listMatch (cond rq) (conditions rl).

Definition rule_conflict: rule -> rule -> Prop := fun r1 r2 =>
  exists rq:request, (rule_permit rq r1 /\ rule_deny rq r2) \/
                     (rule_deny rq r1 /\ rule_permit rq r2).

Definition case1' (m1 M1 m2 M2: Z) : bool :=
  (m1 <=b M2 - MAX) && (M2 - MAX <=b M1).
Definition case2' (m1 M1 m2 M2: Z) : bool :=
  (m1 <=b M2) && (M2 <=b M1) && (m1 <=b m2).
Definition case3' (m1 M1 m2 M2: Z) : bool :=
  (m1 <=b m2) && (m2 <=b M1) && (M1 <=b M2).
```

```
Definition case4' (m1 M1 m2 M2: Z):  bool := M1 <=b M2 - MAX.

Definition allCases (m1 M1 m2 M2: Z)  : bool :=
(case1' m1 M1 m2 M2) || (case2' m1 M1 m2 M2) ||
(case3' m1 M1 m2 M2) ||(case4' m1 M1 m2 M2).

Definition sracCheck: srac -> srac -> bool := fun sr1 sr2 =>
match sr1 with
| any => true
| timeInRange (timeValue m1) (timeValue M1') =>
  match sr2 with
  | any => true
  | timeInRange (timeValue m2) (timeValue M2') =>
    if (m1 <=b m2) then allCases m1 (M'ToM m1 M1') m2 (M'ToM m2 M2')
                   else allCases m2 (M'ToM m2 M2') m1 (M'ToM m1 M1')
  | timeGe (timeValue m2) =>
    if (m1 <=b M1') && (M1' <b m2) then false else true
  | timeLt (timeValue M2) =>
    if (M2 <=b m1) && (m1 <=b M1') then false else true
  | _ => false
  end
| timeGe (timeValue m1) =>
  match sr2 with
  | any => true
  | timeInRange (timeValue m2) (timeValue M2') =>
    if (m2 <=b M2') && (M2' <b m1) then false else true
  | timeGe (timeValue m2) => true
  | timeLt (timeValue M2) => m1 <b M2
  | _ => false
  end
| timeLt (timeValue M1) =>
  match sr2 with
  | any => true
  | timeInRange (timeValue m2) (timeValue M2') =>
    if (M1 <=b m2) && (m2 <=b M2') then false else true
  | timeGe (timeValue m2) => m2 <b M1
  | timeLt (timeValue M2) => true
  | _ => false
  end

| intGt (intValue m1) =>
  match sr2 with
  | any => true
  | intGt (intValue m2) => true
```

```
  | _ => false
  end
end.

Fixpoint listCheck  (l1: list srac)(sr2:srac){struct l1}  : bool :=
  match l1 with
  | nil => false
  | cons sr1 l1' => (sracCheck sr1 sr2) || (listCheck l1' sr2)
  end.

Fixpoint listListCheck  (l1 l2:list srac){struct l2}  : bool :=
  match l2 with
  | nil => false
  | cons sr2 l2' => (listCheck l1 sr2) || (listListCheck l1 l2')
  end.

Definition accessDiff: access -> access -> bool :=
fun a1 a2 =>
match a1, a2 with
  permit, deny => true
| deny, permit  => true
| _, _      => false
end.

Definition conflict_check: rule -> rule -> bool :=
fun r1 r2 =>
match r1 with (ruleCons a1 s1 r1 ac1 c1) =>
match r2 with (ruleCons a2 s2 r2 ac2 c2) =>
  accessDiff a1 a2 &&
  listListCheck s1 s2 &&
  listListCheck r1 r2 &&
  listListCheck ac1 ac2 &&
  listListCheck c1 c2
end
end.

Ltac matchInd H :=
match type of H with
| match ?x with | timeValue _ => _ end = true => induction x
| match ?x with | timeValue _ => _ end => induction x

| match ?x with | intValue _    => _ end = true => induction x
| match ?x with | intValue _    => _ end => induction x
end.
```

```
Ltac matchIndGoal :=
match goal with
| |- match ?x with | timeValue _ => _ end = _ => induction x
| |- _ = match ?x with | timeValue _ => _ end => induction x

| |- match ?x with | intValue _    => _ end = _ => induction x
| |- _ = match ?x with | intValue _    => _ end => induction x
end.

Lemma minInRange: forall (m M : Z), inRange m m M = true.
Proof.
intros.
unfold inRange, inRange'.
repeat simplOrIf; b2P;
first [solve [trivial] | omega | right; b2P; omega].
Qed.

Lemma maxInRange: forall (m M' : Z) (mv M'v : tValue),
mv = timeValue m -> M'v = timeValue M' ->
inRange M' m M' = true.
Proof.
intros.
unfold inRange, inRange'.
generalize (timeConst M'v); rewrite H0.
intro.
test (M' <=b M' = true).
repeat simplOrIf; b2P; tauto.
Qed.

Lemma sracCheckSymetricTimeRanges:
forall (sr1 sr2: srac)(m1v M1'v m2v M2'v : tValue),
sr1 = timeInRange m1v M1'v ->
sr2 = timeInRange m2v M2'v -> sracCheck sr1 sr2 = sracCheck sr2 sr1.
Proof.
intros.

rewrite H, H0.
unfold sracCheck.

generalize (timeConst m1v); generalize (timeConst M1'v);
generalize (timeConst m2v); generalize (timeConst M2'v);
intros.
```

```
remind m1v m1; remind M1'v M1';
remind m2v m2; remind M2'v M2'.

generalize (Ztrichotomy m1 m2); intro.

decompose [or] H5; clear H5.
test (m1 <=b m2);
test (m2 <=b m1);
trivial.

rewrite H7 in *.
destruct H1; destruct H2; destruct H3; destruct H4.
unfold allCases, case1', case2', case3', case4', M'ToM.
test (m2 <=b m2).

caseSimpl (m2<=b M2');
caseSimpl (m2<=b M1');
generalize (pmB M1' MAX); intro; try rewrite H10; clear H10;
generalize (pmB M2' MAX); intro; try rewrite H10; clear H10;
generalize (leSwap M1' M2' MAX); intro; try rewrite H10; clear H10;
generalize (leSwap M2' M1'); intro; try rewrite H10; clear H10;
try rewrite C;
try rewrite C0;
repeat simplOr;
try auto with *.

assert (m2 < m1); [omega| ].
test (m1 <=b m2);
test (m2 <=b m1);
trivial.
Qed.

Hint Resolve sracCheckSymetricTimeRanges.

Lemma sracCheckSymetric: forall (sr1 sr2: srac),
   sracCheck sr1 sr2 = sracCheck sr2 sr1.
Proof.
intros.
remember sr1; induction sr1;
remember sr2; induction sr2;

try (solve[
try (rewrite Heqs, Heqs0);
unfold sracCheck;
```

```
repeat matchIndGoal;
trivial]);

eauto.
Qed.

Lemma rangeSound': forall m1 M1' m2 M2' M1 M2: Z,
0 <= m1 < MAX /\ 0 <= M1' < MAX /\ M1 = M'ToM m1 M1' /\
0 <= m2 < MAX /\ 0 <= M2' < MAX /\ M2 = M'ToM m2 M2' ->
(allCases m1 M1 m2 M2 = true) ->
(exists req : Z, (inBothRanges m1 M1 m2 M2 req) = true).
Proof.
intros.
unfold allCases in H0.
b2PH;
[exists m1 | exists m2 | exists m2 | exists m1];
apply (timeRangeSound _ _ _ _ M1' M2' _);
try tauto;
[do 3 leftb | do 2 leftb; rightb | leftb; rightb | rightb ];
unfold case1', case2', case3', case4' in H0;
unfold case1, case2, case3, case4;
b2P; trivial;
apply minInRange.
Qed.

Lemma rangeSound: forall (m1v M1'v m2v M2'v : tValue) (sr1 sr2: srac),
sr1 = timeInRange m1v M1'v -> sr2 = timeInRange m2v M2'v ->
sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
intros.
unfold sracCheck in H1.
unfold valueMatch.

rewrite H, H0 in *; clear H H0.

generalize (timeConst m1v); generalize (timeConst M1'v);
generalize (timeConst m2v); generalize (timeConst M2'v);
intros.

remind m1v m1; remind M1'v M1';
remind m2v m2; remind M2'v M2'.

define M1 (M'ToM m1 M1').
```

```
define M2 (M'ToM m2 M2').

rewrite <- HeqM1 in H1; rewrite <- HeqM2 in H1.
caseSimpl (m1 <=b m2);
[assert (exists time: Z, inBothRanges m1 M1 m2 M2 time = true) |
 assert (exists time: Z, inBothRanges m2 M2 m1 M1 time = true) ];
[apply (rangeSound' _ M1' _ M2'); tauto | |
 apply (rangeSound' _ M2' _ M1'); tauto | ];
destruct H4;
exists (timeReq (timeValue x));
unfold inBothRanges in H4; b2PH;

generalize (rangeEquiv x m1 M1 M1' H3 H2 HeqM1); intro;
generalize (rangeEquiv x m2 M2 M2' H0 H HeqM2); intro;
rewrite <- H6 , <- H7;
tauto.
Qed.

Lemma rangeGeSound: forall (m1v M1'v m2v : tValue) (sr1 sr2: srac),
sr1 = timeInRange m1v M1'v -> sr2 = timeGe m2v -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
intros.
rewrite H, H0 in *.
unfold sracCheck in H1.
remind m1v m1; remind M1'v M1';
remind m2v m2.
unfold valueMatch, inRange, inRange'.

generalize (timeConst m1v); rewrite Heqm1;
generalize (timeConst M1'v); rewrite HeqM1';
generalize (timeConst m2v); rewrite Heqm2;
intros.

caseSimpl ((m1 <=b M1') && (M1' <b m2)); [exfalso; auto with * | ].
b2PH.

exists (timeReq (timeValue (MAX-1))).
destruct H3; test (M1'<b MAX).
test (m1 <=b M1').
split; [rightb; b2P | ]; omega.

exists (timeReq M1'v).
rewrite HeqM1'.
```

```
split; [ | omega].
apply (maxInRange _ _ m1v M1'v); trivial.
Qed.

Lemma rangeLtSound: forall (m1v M1'v M2v: tValue) (sr1 sr2: srac),
sr1 = timeInRange m1v M1'v -> sr2 = timeLt M2v -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
intros.
rewrite H, H0 in *.
unfold sracCheck in H1.
remind m1v m1; remind M1'v M1';
remind M2v M2.
unfold valueMatch, inRange, inRange'.

generalize (timeConst m1v); rewrite Heqm1;
generalize (timeConst M1'v); rewrite HeqM1';
generalize (timeConst M2v); rewrite HeqM2;
intros.

destruct H3; test (M1' <b MAX).
caseSimpl ((M2 <=b m1) && (m1 <=b M1')); [exfalso; auto with * | ].
b2PH.

exists (timeReq m1v); rewrite Heqm1.
split; [ | omega].
test (m1 <=b m1).
caseSimpl (m1 <=b M1'); trivial.

exists (timeReq (timeValue (-1))).
split; [ | omega].
test (m1 <=b M1').
leftb; b2P.
omega.
Qed.

Lemma geSound: forall (m1v m2v: tValue) (sr1 sr2: srac),
sr1 = timeGe m1v -> sr2 = timeGe m2v -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
intros.
remind m1v m1;
remind m2v m2.
rewrite H, H0.
```

```
unfold valueMatch.
generalize (timeConst m1v); rewrite Heqm1;
generalize (timeConst m2v); rewrite Heqm2.
intros.

exists (timeReq (timeValue (MAX-1))).
omega.
Qed.

Lemma geLtSound: forall (m1v M2v: tValue) (sr1 sr2: srac),
sr1 = timeGe m1v -> sr2 = timeLt M2v -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
intros.
remind m1v m1;
remind M2v M2.
rewrite H, H0 in *.
unfold sracCheck in H1.
b2PH.
exists (timeReq (timeValue m1)).
simpl.
omega.
Qed.

Lemma ltSound: forall (M1v M2v: tValue) (sr1 sr2: srac),
sr1 = timeLt M1v -> sr2 = timeLt M2v -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
intros.
remind M1v M1;
remind M2v M2.
rewrite H, H0.
generalize (timeConst M1v); rewrite HeqM1;
generalize (timeConst M2v); rewrite HeqM2;
exists (timeReq (timeValue (-1))).
simpl.
omega.
Qed.

Lemma intGtSound: forall (m1v m2v: iValue) (sr1 sr2: srac),
sr1 = intGt m1v -> sr2 = intGt m2v -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
intros.
```

```
remind m1v m1;
remind m2v m2.
rewrite H, H0.

caseSimpl (m1 <=b m2);
[ exists (intReq(intValue (m2+1))) | exists (intReq(intValue (m1+1))) ];
simpl;
b2PH;
auto with *.
Qed.

Lemma anySound: forall (sr1 sr2: srac),
sr1 = any -> sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
intros.
generalize (hasOneMatch sr2); intro.
destruct H1.
exists x.
split.

rewrite H.
unfold valueMatch.
induction x;
[induction t | induction i];
trivial.
trivial.
Qed.

Hint Resolve rangeGeSound rangeSound geSound rangeLtSound
             geLtSound ltSound intGtSound anySound.

Ltac elimDiffTypes H Heqs Heqs0:=
rewrite Heqs, Heqs0 in H;
unfold sracCheck in H;
repeat matchInd H;
exfalso;
auto with *.

Lemma existAndCommute : forall (sr1 sr2: srac),
(exists reqv: reqValue, valueMatch reqv sr1 /\ valueMatch reqv sr2) ->
 exists reqv: reqValue, valueMatch reqv sr2 /\ valueMatch reqv sr1.
Proof.
intros.
```

```
destruct H.
exists x.
tauto.
Qed.

Ltac soundSwap s s0 :=
apply (existAndCommute);
replace (sracCheck s s0) with (sracCheck s0 s) in *;
[| apply (sracCheckSymetric s0 s)].

Ltac swapIfSmallerSound Heqs Heqs0 s s0 C0 :=
caseSimpl ((rank s0) <b (rank s));
[soundSwap s s0 | ];
rewrite Heqs, Heqs0 in C0;
unfold rank in C0;
try solve [exfalso; b2PH; omega].

Lemma sracCheckSoundness: forall sr1 sr2: srac,
 sracCheck sr1 sr2 = true ->
exists reqv: reqValue, (valueMatch reqv sr1) /\ (valueMatch reqv sr2).
Proof.
intros.
remember sr1; induction sr1;
remember sr2; induction sr2;
try (solve [elimDiffTypes H Heqs Heqs0]);
swapIfSmallerSound Heqs Heqs0 s s0 C;
eauto.
Qed.

Lemma listCheckSoundness: forall l1 l2: list srac,
listListCheck l1 l2 = true ->
exists reqv: reqValue, (listMatch reqv l1) /\ (listMatch reqv l2).
Proof.
intros.
unfold listListCheck in H.
induction l2.
exfalso; auto with *.
b2PH.
clear IHl2.

unfold listCheck in H.
induction l1.
exfalso; auto with *.
b2PH.
```

```
clear IHl1.

generalize (sracCheckSoundness _ _ H); intro.
destruct H0; exists x; simpl; tauto.

fold listCheck in *.
generalize (IHl1 H); intros.
destruct H0; exists x; simpl; tauto.

fold listListCheck in *.
generalize (IHl2 H); intros.
destruct H0; exists x; simpl; tauto.
Qed.

Lemma conflict_check_soundness:
  forall r1 r2: rule,
    conflict_check r1 r2  = true -> rule_conflict r1 r2.
Proof.
intros [a1 s1 re1 ac1 c1] [a2 s2 re2 ac2 c2].
unfold rule_conflict, rule_permit, rule_deny;
simpl.
intros.
b2PH.

generalize (listCheckSoundness _ _ H0); intro; destruct H4.
generalize (listCheckSoundness _ _ H1); intro; destruct H5.
generalize (listCheckSoundness _ _ H2); intro; destruct H6.
generalize (listCheckSoundness _ _ H3); intro; destruct H7.
clear H0 H1 H2 H3.
induction a1; induction a2;
unfold accessDiff in H;
try (solve [exfalso; auto with *]);

exists (requestCons x2 x1 x0 x);
[left | right];
tauto.
Qed.

(* Completeness *)
Lemma rangeComp' : forall m1 M1 m2 M2 M1' M2' req : Z,
inRange req m1 M1 = true -> inRange req m2 M2 = true -> m1 <= m2 ->
0 <= req < MAX /\
0 <= m1 < MAX /\ 0 <= M1' < MAX /\ M1 = M'ToM m1 M1' /\
0 <= m2 < MAX /\ 0 <= M2' < MAX /\ M2 = M'ToM m2 M2' ->
```

```
allCases m1 M1 m2 M2 = true.
Proof.
intros.

assert (inBothRanges m1 M1 m2 M2 req = true).
unfold inBothRanges.
auto with *.

generalize (timeRangeComp m1 M1 m2 M2 M1' M2' req H2 H1 H3); intros.
clear - H4;
unfold allCases, case1, case2, case3, case4,
                 case1', case2', case3', case4' in *.

destruct H4.
do 3 leftb; b2P; trivial.
destruct H. do 2 leftb; rightb; b2P; trivial.
destruct H. leftb; rightb; b2P; trivial.
rightb; b2P; trivial.
Qed.

Lemma rangeComp: forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(m1v M1'v m2v M2'v : tValue),
sr1 = timeInRange m1v M1'v -> sr2 = timeInRange m2v M2'v ->
 (valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.
Proof.
intros.
unfold valueMatch in *.
unfold sracCheck.

generalize (timeConst m1v); generalize (timeConst M1'v);
generalize (timeConst m2v); generalize (timeConst M2'v);
generalize (timeConst reqv);
rewrite H, H0, H3 in *.
intros.

remind m1v m1; remind M1'v M1';
remind m2v m2; remind M2'v M2';

define M1 (M'ToM m1 M1').
define M2 (M'ToM m2 M2').

assert ( inRange req m1 M1 = inRange req m1 M1').
apply rangeEquiv; trivial.
```

```
assert ( inRange req m2 M2 = inRange req m2 M2').
apply rangeEquiv; trivial.

rewrite <- H9 in H1.
rewrite <- H10 in H2.

assert (inBothRanges m1  M1 m2 M2 req = true).
unfold inBothRanges; b2P; trivial.

assert (inBothRanges m2  M2 m1 M1 req = true).
unfold inBothRanges; b2P; trivial.

caseSimpl (m1 <=b m2); b2PH.
apply (rangeComp' _ _ _ _ M1' M2' req); tauto.
apply (rangeComp' _ _ _ _ M2' M1' req); try tauto.
omega.
Qed.

Lemma rangeGeComp: forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(m1v M1'v m2v : tValue),
sr1 = timeInRange m1v M1'v -> sr2 = timeGe m2v ->
(valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.
Proof.
intros.
rewrite H, H0 in *.
unfold valueMatch in H1, H2.

remind m1v m1; remind M1'v M1';
remind m2v m2.
rewrite H3 in H1, H2.

unfold sracCheck.
unfold inRange in H1.

generalize (timeConst m1v); generalize (timeConst M1'v);
generalize (timeConst m2v); generalize (timeConst reqv);
rewrite Heqm1, HeqM1', Heqm2, H3;
intros.

destruct H6; test (M1' <b MAX).
unfold inRange' in H1.
```

```
caseSimpl (m1 <=b M1'); [ | trivial].
caseSimpl (M1' <b m2); [ | trivial].
b2P.
exfalso.
omega.
Qed.

Lemma rangeLtComp: forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(m1v M1'v M2v : tValue),
sr1 = timeInRange m1v M1'v -> sr2 = timeLt M2v ->
(valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.
Proof.
intros.
rewrite H, H0 in *.
unfold valueMatch in H1, H2.

remind m1v m1; remind M1'v M1';
remind M2v M2.
rewrite H3 in H1, H2.

unfold sracCheck.
unfold inRange in H1.

generalize (timeConst m1v); generalize (timeConst M1'v);
generalize (timeConst M2v); generalize (timeConst reqv);
rewrite Heqm1, HeqM1', HeqM2, H3;
intros.

destruct H6; test (M1' <b MAX).
unfold inRange' in H1.

caseSimpl (m1 <=b M1'); [ | caseSimpl (M2 <=b m1); trivial].
b2P.
assert (m1 < M2); [omega | ].
test (M2 <=b m1).
trivial.
Qed.

Lemma geComp:  forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(m1v m2v : tValue),
sr1 = timeGe m1v -> sr2 = timeGe m2v ->
 (valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.
```

```
Proof.
intros.
rewrite H, H0 in *.
simpl.
induction m1v; induction m2v.
trivial.
Qed.

Lemma geLtComp:  forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(m1v M2v : tValue),
sr1 = timeGe m1v -> sr2 = timeLt M2v ->
(valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.
Proof.
intros.
rewrite H, H0 in *.
simpl.
remind m1v m1; remind M2v M2.
rewrite H3 in H1, H2.
unfold valueMatch in H1, H2.
b2P.
omega.
Qed.

Lemma ltComp:  forall (reqv : tValue) (req : Z) (sr1 sr2: srac)
(M1v M2v : tValue),
sr1 = timeLt M1v -> sr2 = timeLt M2v ->
(valueMatch (timeReq reqv) sr1) -> (valueMatch (timeReq reqv) sr2) ->
reqv = timeValue req -> sracCheck sr1 sr2 = true.
Proof.
intros.
rewrite H, H0 in *.
simpl.
induction M1v; induction M2v.
trivial.
Qed.

Lemma intGtComp:  forall (reqv : iValue) (req : Z) (sr1 sr2: srac)
(m1v m2v : iValue),
sr1 = intGt m1v -> sr2 = intGt m2v ->
(valueMatch (intReq reqv) sr1) -> (valueMatch (intReq reqv) sr2) ->
reqv = intValue req -> sracCheck sr1 sr2 = true.
Proof.
intros.
```

```
rewrite H, H0 in *.
simpl.
induction m1v; induction m2v.
trivial.
Qed.

Lemma anyComp: forall (reqv : reqValue) (sr1 sr2: srac),
sr1 = any -> sracCheck sr1 sr2 = true.
Proof.
intros.
rewrite H.
unfold sracCheck.
trivial.
Qed.

Hint Resolve rangeComp rangeGeComp geComp rangeLtComp
             geLtComp ltComp intGtComp anyComp.

Ltac diffTypesComp H Heqs :=
unfold valueMatch in H;
rewrite Heqs in H;
repeat matchInd H;
tauto.

Ltac compSwap s s0 :=
replace (sracCheck s s0) with (sracCheck s0 s);
[| apply (sracCheckSymetric s0 s)].

Ltac swapIfSmallerComp Heqs Heqs0 s s0 C0 :=
caseSimpl ((rank s0) <b (rank s));
[compSwap s s0 | ];
rewrite Heqs, Heqs0 in C0;
unfold rank in C0;
try solve [exfalso; b2PH; omega].

Lemma sracCompleteness: forall (reqv : reqValue) (sr1 sr2: srac),
 (valueMatch reqv sr1) -> (valueMatch reqv sr2) ->
sracCheck sr1 sr2 = true.
Proof.
intros.
remember reqv; induction reqv; rewrite Heqr in *;

match type of Heqr with
| _ = _ ?t => induction t
```

```
end;

remember sr1; induction sr1;
remember sr2; induction sr2;
try (solve [diffTypesComp H Heqs]);
try (solve [diffTypesComp H0 Heqs0]);
swapIfSmallerComp Heqs Heqs0 s s0 C;
eauto.
Qed.

Lemma listCompleteness: forall (reqv: reqValue) (l1 l2:list srac),
(listMatch reqv l1) -> (listMatch reqv l2) -> (listListCheck l1 l2 = true).
Proof.
intros.

induction l2.
unfold listMatch in H0.
destruct H0.

unfold listMatch in H0; fold listMatch in H0.
simpl.
destruct H0.

leftb.
clear IHl2.
induction l1.
unfold listMatch in H.
destruct H.

simpl.
unfold listMatch in H; fold listMatch in H.
destruct H.

leftb; apply (sracCompleteness reqv); trivial.
rightb; apply IHl1; trivial.
rightb; apply IHl2; trivial.
Qed.

Lemma conflict_check_completeness:
  forall r1 r2: rule,
     rule_conflict r1 r2 -> (conflict_check r1 r2 = true).
Proof.
intros [a1 s1 re1 ac1 c1] [a2 s2 re2 ac2 c2].
unfold rule_conflict, rule_permit, rule_deny.
```

```
simpl.
intros.
elim H; intros; clear H.
decompose [and or] H0; clear H0;

generalize (listCompleteness _ _ _ H1 H2); intro;
generalize (listCompleteness _ _ _ H3 H7); intro;
generalize (listCompleteness _ _ _ H4 H8); intro;
generalize (listCompleteness _ _ _ H6 H10); intro;
rewrite H, H5, H0, H9, H11, H12; trivial.
Qed.

End conflict.
```

Appendix D

RuleSet.v

In this appendix, we extend the algorithm to work with rule sets instead of just rules. We also prove the correctness of this extension. The entirety of this code comes from another paper [3]. It has only been modified to use the definitions and lemmas from the XACML rules as opposed to the firewall rules it has been designed for. The same proof commands are used to prove correctness of both algorithms (the conflict detection on firewall rules algorithm and the XACML rule conflicts algorithm). Only five lines were modified for this code to work with our algorithm, those being:

```
Require Export ConflictDetect.
```

We add this line in order to use the XACML rules and theorems. We also removed the parts from the original code that has been redefined.

```
Hypothesis timeConst: forall v : tValue,
match v with
  | timeValue v' => 0 <= v' < MAX
end.

Hypothesis hasOneMatch: forall sr : srac,
  exists reqv: reqValue, valueMatch reqv sr.
```

We add our hypothesis on time values and the one that all rules satisfy at least one request.

```
Lemma rrs_conflicts_soundness:
...
generalize (conflict_check_soundness timeConst hasOneMatch r r0).

Lemma conflicts_0_Sj_rrs_conflicts:
...
rewrite (conflict_check_completeness timeConst).
```

In two lemmas, we modify commands that use XACML conflict lemmas that need the time hypothesis. We simply add the hypothesis as an argument.

The rest of the code (seen bellow) is taken directly from the firewall paper.

```
Require Export ConflictDetect.

Hypothesis timeConst: forall v : tValue,
match v with
  | timeValue v' => 0 <= v' < MAX
end.

Hypothesis hasOneMatch: forall sr : srac,
  exists reqv: reqValue, valueMatch reqv sr.

Definition rule_set := list rule.

(* Projection on lists, without default value. *)
Fixpoint Nth (A:Set)(l:list A){struct l}:
  forall i:nat, (i<length l)%nat -> A :=
match l return forall i:nat, (i<length l)%nat -> A with
  nil => fun i H => False_rec A (lt_n_O i H)
| cons a l => fun i =>
     match i return (i<length (cons a l))%nat -> A with
       O%nat => fun H => a
     | S i => fun H => Nth A l i (lt_S_n i (length l) H)
     end
end.

Implicit Arguments Nth [A].

Lemma Nth_proof_independent:
  forall (A:Set)(l:list A)(i:nat)(H1 H2:(i<length l)%nat),
    Nth l i H1 = Nth l i H2.
Proof.
```

```
intros A l; elim l; simpl.
intros i H1; elim (lt_n_O i H1).
intros a l' IH i; case i.
trivial.
intros i' H1 H2; apply IH.
Qed.

(* Dependent conjunction. *)
Inductive dep_and (A:Prop)(B:A->Prop): Prop :=
  dep_pair: forall (a:A)(b:B a), dep_and A B.

Notation " { A ** B } " := (dep_and A B).


(* Some small lemmas about order on natural numbers
   not provided by the library. *)
Lemma lt_O_lt_minus:
  forall n m:nat, lt n m -> lt 0 (m-n).
Proof.
intros n m H.
apply plus_lt_reg_l with n.
rewrite plus_O_r.
rewrite le_plus_minus_r.
exact H.
apply lt_le_weak.
exact H.
Qed.

Lemma minus_S_pred: forall n m:nat, (n-(S m))%nat = pred (n-m).
Proof.
induction n; simpl.
trivial.
intros m; case m; simpl.
symmetry; apply minus_n_O.
exact IHn.
Qed.


(* Finds conflicts between a rule and a rule set:
     i = position of the rule r in the original rule set
     n = position of the first rule of rs in the original rule set *)
Fixpoint rrs_conflicts (i n:nat)(r:rule)(rs:rule_set){struct rs}:
  list (nat*nat) :=
match rs with
```

```
  nil => nil
| (cons rh rt) => if (conflict_check r rh)
                  then (i,n)::(rrs_conflicts i (S n) r rt)
                  else (rrs_conflicts i (S n) r rt)
end.

(* Find conflicts (assuming that rs starts with rule number n *)
Fixpoint conflicts_aux (n:nat)(rs:rule_set){struct rs}:
  list (nat*nat) :=
match rs with
  nil => nil
| (cons r rs') => (rrs_conflicts n (S n) r rs') ++ (conflicts_aux (S n) rs')
end.

(* The conflict-detection program. *)
Definition find_conflicts: rule_set -> list (nat*nat)
:= conflicts_aux 0.

(* rrs_conflict_res r rs n means that there is a conflict between
   r and the nth rule of rs *)
Inductive rrs_conflict_rel (r:rule): rule_set -> nat -> Prop :=
  rrs_conflict_head:
    forall (rh:rule)(rt:rule_set),
      rule_conflict r rh -> rrs_conflict_rel r (rh::rt) 0
| rrs_conflict_tail:
    forall (rh:rule)(rt:rule_set)(n:nat),
      rrs_conflict_rel r rt n -> rrs_conflict_rel r (rh::rt) (S n).

Lemma rrs_conflicts_n_le_j:
  forall (i i':nat)(r:rule)(rs:rule_set)(j n:nat),
    In (i,j) (rrs_conflicts i' n r rs) -> (n<=j)%nat.
Proof.
intros i i' r rs; elim rs; simpl.
intros j n H; elim H.
intros r0 rs' IH j n.
case (conflict_check r r0); simpl.
intros H; case H; clear H; intro H.
injection H; clear H; intros Hi Hj.
rewrite Hi; apply le_n.
apply le_Sn_le; apply IH; exact H.
intro H; apply le_Sn_le; apply IH; exact H.
Qed.

(* Soundness *)
```

```
Lemma rrs_conflicts_soundness:
  forall (i i':nat)(r:rule)(rs:rule_set)(j n:nat),
    In (i,j) (rrs_conflicts i' n r rs) ->
    i'=i /\ rrs_conflict_rel r rs (j-n)%nat.
Proof.
intros i i' r rs; elim rs; simpl.
intros j n H; elim H.
intros r0 rs' IH j n.
generalize (conflict_check_soundness timeConst hasOneMatch r r0).
case (conflict_check r r0); simpl.
intros H0 H; case H; clear H; intro H.
injection H; clear H; intros H1 H2.
split.
exact H2.
rewrite H1; rewrite <- minus_n_n.
apply rrs_conflict_head.
apply H0; trivial.
generalize (IH j (S n) H); intros [IH1 IH2].
split.
exact IH1.
rewrite (S_pred (j-n) 0).
apply rrs_conflict_tail.
rewrite <- minus_S_pred.
exact IH2.
apply lt_0_lt_minus.
red; eapply rrs_conflicts_n_le_j; apply H.

intros _ H; generalize (IH j (S n) H); intros [IH1 IH2].
split.
exact IH1.
rewrite (S_pred (j-n) 0).
apply rrs_conflict_tail.
rewrite <- minus_S_pred.
exact IH2.
apply lt_0_lt_minus.
red; eapply rrs_conflicts_n_le_j; apply H.
Qed.

Definition rs_conflict: rule_set -> nat -> nat -> Prop :=
fun rs i j =>
  {(i < (length rs))%nat ** fun Hi =>
  {(j < (length rs))%nat ** fun Hj =>
   (rule_conflict (Nth rs i Hi) (Nth rs j Hj))}}.
```

```
Lemma rrs_conflict_rel_rs_conflict_aux:
  forall (r:rule)(rs:rule_set)(m:nat), rrs_conflict_rel r rs m ->
    forall n:nat, (n = S m) -> rs_conflict (r::rs) 0 n.
Proof.
intros r rs m H; elim H.
intros rh rt Hr n Hn1.
rewrite Hn1; red.
cut (0<S (length (rh::rt)))%nat.
intro HH1; exists HH1.
cut (1<S (S (length rt)))%nat.
intro HH2; exists HH2.
exact Hr.
apply lt_n_S; apply lt_O_Sn.
apply lt_O_Sn.

intros rh rt m' Hm' IH n Hn.
rewrite Hn.
generalize (IH (S m') (refl_equal (S m'))); intros [H1 [H2 H3]].
red.
exists (lt_O_Sn (length (rh::rt))).
exists (lt_n_S _ _ H2).
simpl; simpl in H3.
rewrite Nth_proof_independent with (H2:=lt_S_n m' (length rt) H2).
exact H3.
Qed.

Lemma rrs_conflict_rel_rs_conflict:
  forall (r:rule)(rs:rule_set)(n:nat),
    (0<n)%nat -> rrs_conflict_rel r rs (pred n) -> rs_conflict (r::rs) 0 n.
Proof.
intros r rs n Hn H; apply rrs_conflict_rel_rs_conflict_aux with (m:=pred n).
exact H.
apply S_pred with (m:=0%nat).
exact Hn.
Qed.

Lemma rs_conflict_cons:
  forall (r:rule)(rs:rule_set)(i j:nat),
    rs_conflict rs i j -> rs_conflict (r::rs) (S i) (S j).
Proof.
intros r rs i j [H1 [H2 H3]]; red.
exists (lt_n_S _ _ H1).
exists (lt_n_S _ _ H2).
simpl.
```

```
rewrite Nth_proof_independent with (H2:=H1).
rewrite Nth_proof_independent with (H2:=H2).
exact H3.
Qed.

Lemma conflicts_aux_le:
  forall (rs:rule_set)(i j n:nat),
    In (i,j) (conflicts_aux n rs) -> (n<=i)%nat /\ (n<=j)%nat.
Proof.
intro rs; elim rs; simpl.
intros i j n H; elim H.
clear rs; intros r rs IH i j n H.
case in_app_or with (1:=H); clear H; intro H.
split.
generalize (rrs_conflicts_soundness _ _ _ _ _ _ H); intros [Hi _].
rewrite Hi; apply le_n.
apply lt_le_weak.
apply rrs_conflicts_n_le_j with (1:=H).

generalize (IH _ _ _ H); intros [H1 H2].
split; apply lt_le_weak; assumption.
Qed.

Lemma conflicts_aux_soundness:
  forall (rs:rule_set)(n i j:nat),
    In (i,j) (conflicts_aux n rs) -> rs_conflict rs (i-n)%nat (j-n)%nat.
Proof.
intros rs; elim rs; simpl.
intros n i j H; elim H.
intros r rs' IH n i j H.
case in_app_or with (1:=H); clear H; intro H.
generalize (rrs_conflicts_soundness i n r rs' j (S n) H); intros [Hi HH].
rewrite <- Hi; rewrite <- minus_n_n.
apply rrs_conflict_rel_rs_conflict.
apply lt_0_lt_minus.
red.
apply rrs_conflicts_n_le_j with (1:=H).
rewrite <- minus_S_pred.
exact HH.
generalize (conflicts_aux_le _ _ _ _ H); intros [H1 H2].
rewrite (S_pred (i-n) 0).
rewrite (S_pred (j-n) 0).
apply rs_conflict_cons.
rewrite <- (minus_S_pred i n); rewrite <- (minus_S_pred j n).
```

```
apply IH; exact H.
apply lt_0_lt_minus; exact H2.
apply lt_0_lt_minus; exact H1.
Qed.

Theorem conflicts_soundness:
  forall (rs:rule_set)(i j:nat),
    In (i,j) (find_conflicts rs) -> rs_conflict rs i j.
Proof.
intros rs i j H.
rewrite (minus_n_O i); rewrite (minus_n_O j).
apply conflicts_aux_soundness.
exact H.
Qed.



(* Completeness *)
Lemma Nth_singleton:
  forall (A:Set)(a:A)(i:nat)(hi:(i<1)%nat),
    Nth (a::nil) i hi = a.
Proof.
intros A a i; case i; simpl.
trivial.
intros i' hi'.
red in hi'.
elim (le_Sn_O i').
apply le_S_n; exact hi'.
Qed.

Lemma rule_permit_no_deny:
  forall (rl:rule)(r:request),
    rule_permit r rl -> rule_deny r rl -> False.
Proof.
intros rl r [H1 _] [H2 _].
rewrite H1 in H2.
discriminate H2.
Qed.

Lemma no_rule_conflict_self:
  forall r:rule, ~ (rule_conflict r r).
Proof.
intros r [q H]; case H; intros [H1 H2];
apply rule_permit_no_deny with r q; assumption.
```

```
Qed.

Lemma rs_conflict_singleton:
  forall (r:rule)(i j:nat),
    ~ (rs_conflict (r::nil) 0 j).
Proof.
intros r i j [h1 [h2 H]].
rewrite Nth_singleton in H.
rewrite Nth_singleton in H.
apply no_rule_conflict_self with r; exact H.
Qed.

Lemma rrs_conflicts_S:
  forall (r:rule)(rs:rule_set)(n j m:nat),
    In (n, j) (rrs_conflicts n m r rs) ->
    In (n, S j) (rrs_conflicts n (S m) r rs).
Proof.
intros r rs; elim rs; simpl.
trivial.
clear rs; intros r1 rs IH n j m; case (conflict_check r r1).
intro H; case H.
intro Heq.
replace m with (snd (n,m)).
rewrite Heq; simpl.
left; trivial.
trivial.

intro H'.
right.
apply IH.
exact H'.
apply IH.
Qed.

Lemma conflicts_0_Sj_rrs_conflicts:
  forall (n:nat)(r:rule)(rs:rule_set)(j:nat),
  rs_conflict (r::rs) 0 (S j) ->
  In (n,j+(S n))%nat (rrs_conflicts n (S n) r rs).
Proof.
intros n r rs j; generalize rs; clear rs; elim j.
intro rs; case rs.
intro H; elim (rs_conflict_singleton r 0 1 H).
clear rs; intros r1 rs.
intros [h1 [h2 H]].
```

```
simpl in H.
simpl.
rewrite (conflict_check_completeness timeConst).
simpl.
left; trivial.
exact H.

clear j; intros j IH.
intro rs; case rs.
intro H; elim (rs_conflict_singleton r 0 (S (S j)) H).
clear rs; intros r1 rs H.
simpl.
cut (In (n, S (j+S n)) (rrs_conflicts n (S (S n)) r rs)).
intro HH.
case (conflict_check r r1).
right.
exact HH.
exact HH.
apply rrs_conflicts_S.
apply IH.

case H; clear H; intros h1 [h2 H].
simpl in h1; simpl in h2; simpl in H.

exists (lt_O_Sn (length rs)).
esplit.
simpl.
apply H.
Qed.

Lemma conflicts_0_rrs_conflicts:
  forall (n:nat)(r:rule)(rs:rule_set)(j:nat),
  (0<j)%nat ->
  rs_conflict (r::rs) 0 j ->
  In (n,j+n)%nat (rrs_conflicts n (S n) r rs).
Proof.
intros n r rs j Hj; rewrite S_pred with (m:=0%nat)(1:=Hj).
intro H.
rewrite plus_Sn_m; rewrite plus_n_Sm.
apply conflicts_0_Sj_rrs_conflicts.
exact H.
Qed.

Lemma rs_conflict_S:
```

```
  forall (r:rule)(rs:rule_set)(i j:nat),
    rs_conflict (r::rs) (S i) (S j) -> rs_conflict rs i j.
Proof.
intros r rs i j [Hi [Hj H]].
simpl in Hi; simpl in Hj; simpl in H.
esplit.
esplit.
apply H.
Qed.

Lemma conflicts_aux_completeness:
  forall (i j:nat), (i<j)%nat ->
  forall (n:nat)(rs:rule_set),
    rs_conflict rs i j -> In (i+n,j+n)%nat (conflicts_aux n rs).
Proof.
intro i; elim i.
intros j h n rs; case rs.
intros [Hi H].
elim lt_n_O with O.
exact Hi.

clear rs; intros r rs H; simpl.
apply in_or_app.
left.
apply conflicts_0_rrs_conflicts.
exact h.
exact H.

clear i; intros i IH j h.
rewrite (S_pred j (S i) h).
intros n rs; case rs.
intros [Hi H].
elim lt_n_O with (S i).
exact Hi.

clear rs; intros r rs Hj; simpl.
apply in_or_app; right.
rewrite plus_n_Sm with i n.
rewrite plus_n_Sm with (pred j) n.
apply IH with (j:=pred j)(n:=S n).
apply lt_pred.
exact h.
apply rs_conflict_S with r.
exact Hj.
```

```
Qed.

Theorem conflicts_completeness:
  forall (rs:rule_set)(i j:nat), (i<j)%nat ->
    rs_conflict rs i j -> In (i,j) (find_conflicts rs).
Proof.
intros rs i j Hlt H; unfold find_conflicts; simpl.
rewrite <- plus_0_r with (n:=i); rewrite <- plus_0_r with (n:=j).
apply conflicts_aux_completeness.
exact Hlt.
exact H.
Qed.
```

Appendix E

Example Policy

In this Appendix, we present an example XACML policy. We created this policy in the testing phase for finding intersections between two time ranges. For readability, we will replace function names with shortened versions. We use the following four:

```
time-in-range,
time-one-and-only,
time,
current-time
```

instead of

```
http://research.sun.com/projects/xacml/names/function#time-in-range,
urn:oasis:names:tc:xacml:1.0:function:time-one-and-only,
http://www.w3.org/2001/XMLSchema#time,
urn:oasis:names:tc:xacml:1.0:environment:current-time
```

The rules in this policy file only use the XACML function *time-in-range*. The rules either permit or deny requests (based on the *Effect* field in the rule) as long as the current time is in the time range found in each RuleID field (based on attribute values found in the rule).

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="urn:oasis:names:tc:xacml:1.0:policy"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      PolicyId="TimeRangePolicy"
      RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:
        rule-combining-algorithm:ordered-permit-overrides">

  <Target>
    <Subjects>
      <anySubject/>
    </Subjects>

    <Resources>
      <anyResource/>
    </Resources>

    <Actions>
      <anyAction/>
    </Actions>
  </Target>

  <Rule RuleId="Permit 9-17" Effect="Permit">
    <Condition FunctionId="time-in-range">
      <Apply FunctionId="time-one-and-only">
        <EnvironmentAttributeDesignator DataType="time"
                                        AttributeId="current-time"/>
      </Apply>
      <AttributeValue DataType="time">09:00:00</AttributeValue>
      <AttributeValue DataType="time">17:00:00</AttributeValue>
    </Condition>
  </Rule>

  <Rule RuleId="Deny 8-12" Effect="Deny">
    <Condition FunctionId="time-in-range">
      <Apply FunctionId="time-one-and-only">
        <EnvironmentAttributeDesignator DataType="time"
                                        AttributeId="current-time"/>
      </Apply>
      <AttributeValue DataType="time">08:00:00</AttributeValue>
      <AttributeValue DataType="time">12:00:00</AttributeValue>
    </Condition>
  </Rule>

  <Rule RuleId="Deny 10-12" Effect="Deny">
```

```
    <Condition FunctionId="time-in-range">
      <Apply FunctionId="time-one-and-only">
        <EnvironmentAttributeDesignator DataType="time"
                                        AttributeId="current-time"/>
      </Apply>
      <AttributeValue DataType="time">10:00:00</AttributeValue>
      <AttributeValue DataType="time">12:00:00</AttributeValue>
    </Condition>
  </Rule>

  <Rule RuleId="Deny 19-8" Effect="Deny">
    <Condition FunctionId="time-in-range">
      <Apply FunctionId="time-one-and-only">
        <EnvironmentAttributeDesignator DataType="time"
                                        AttributeId="current-time"/>
      </Apply>
      <AttributeValue DataType="time">19:00:00</AttributeValue>
      <AttributeValue DataType="time">08:00:00</AttributeValue>
    </Condition>
  </Rule>

  <Rule RuleId="Deny 15-20" Effect="Deny">
    <Condition FunctionId="time-in-range">
      <Apply FunctionId="time-one-and-only">
        <EnvironmentAttributeDesignator DataType="time"
                                        AttributeId="current-time"/>
      </Apply>
      <AttributeValue DataType="time">15:00:00</AttributeValue>
      <AttributeValue DataType="time">20:00:00</AttributeValue>
    </Condition>
  </Rule>

  <Rule RuleId="Deny 20-15" Effect="Deny">
    <Condition FunctionId="time-in-range">
      <Apply FunctionId="time-one-and-only">
        <EnvironmentAttributeDesignator DataType="time"
                                        AttributeId="current-time"/>
      </Apply>
      <AttributeValue DataType="time">20:00:00</AttributeValue>
      <AttributeValue DataType="time">15:00:00</AttributeValue>
    </Condition>
```

```
  </Rule>

  <Rule RuleId="Deny 16-12" Effect="Deny">
    <Condition FunctionId="time-in-range">
      <Apply FunctionId="time-one-and-only">
        <EnvironmentAttributeDesignator DataType="time"
                                        AttributeId="current-time"/>
      </Apply>
      <AttributeValue DataType="time">16:00:00</AttributeValue>
      <AttributeValue DataType="time">12:00:00</AttributeValue>
    </Condition>
  </Rule>

  <Rule RuleId="Deny 22-20" Effect="Deny">
    <Condition FunctionId="time-in-range">
      <Apply FunctionId="time-one-and-only">
        <EnvironmentAttributeDesignator DataType="time"
                                        AttributeId="current-time"/>
      </Apply>
      <AttributeValue DataType="time">22:00:00</AttributeValue>
      <AttributeValue DataType="time">20:00:00</AttributeValue>
    </Condition>
  </Rule>

  <Rule RuleId="Permit 8-10" Effect="Permit">
    <Condition FunctionId="time-in-range">
      <Apply FunctionId="time-one-and-only">
        <EnvironmentAttributeDesignator DataType="time"
                                        AttributeId="current-time"/>
      </Apply>
      <AttributeValue DataType="time">08:00:00</AttributeValue>
      <AttributeValue DataType="time">10:00:00</AttributeValue>
    </Condition>
  </Rule>

  <Rule RuleId="DenyAllOthers" Effect="Deny"/>
</Policy>
```

Appendix F

Testing the Algorithm

In this appendix, we describe the execution of the Java version of our algorithm. This version of the algorithm, while not verified, closely corresponds to the verified algorithm presented in Coq. It will be used on the example policy found in Appendix E. The Java algorithm uses the same approach in finding conflicts between rules (such as the four cases for finding conflicts between two time ranges, see Figure 4.3).

The biggest difference between the two algorithms is how it returns the conflicts. In the Coq algorithm, we simply return the pairs of indices which conflict. In the Java version, we return that list, and also uses each rule's names (each XACML rule is named by the policy writer) as well as the intersection between the rules (i.e. which requests are conflicts). We get these intersections from the last conjunction in each $case_i$ found on page 42.

Below is an example of what the program returns when it finds a conflict:

```
(0, 1): "Permit 9-17" and "Deny 8-12": target[];
  conditions[time-in-range; value[time; 9:0:0]; value[time; 12:0:0]]
```

This says that the first and second rules (rule 0 and 1) conflict. They are named "Permit 9-17" and "Deny 8-12". Their intersection is any target (denoted by empty square bracket), but has a condition in which the current time must be in the range

9 to 12. When finding conflicts between rule 0 and rule 6, the intersection will be two distinct ranges. Each of these ranges is written separated by a comma.

Below is the complete list of conflicts the algorithm finds when run on the sample policy.

```
Conflicts in "time-range.xml"
(0, 1): "Permit 9-17" and "Deny 8-12": target[];
  conditions[time-in-range; value[time; 9:0:0]; value[time; 12:0:0]]

(0, 2): "Permit 9-17" and "Deny 10-12": target[];
  conditions[time-in-range; value[time; 10:0:0]; value[time; 12:0:0]]

(0, 4): "Permit 9-17" and "Deny 15-20": target[];
  conditions[time-in-range; value[time; 15:0:0]; value[time; 17:0:0]]

(0, 5): "Permit 9-17" and "Deny 20-15": target[];
  conditions[time-in-range; value[time; 9:0:0]; value[time; 15:0:0]]

(0, 6): "Permit 9-17" and "Deny 16-12": target[];
  conditions[time-in-range; value[time; 9:0:0]; value[time; 12:0:0],
             time-in-range; value[time; 16:0:0]; value[time; 17:0:0]]

(0, 7): "Permit 9-17" and "Deny 22-20": target[];
  conditions[time-in-range; value[time; 9:0:0]; value[time; 17:0:0]]

(0, 9): "Permit 9-17" and "DenyAllOthers": target[];
  conditions[time-in-range; value[time; 9:0:0]; value[time; 17:0:0]]

(1, 8): "Deny 8-12" and "Permit 8-10": target[];
  conditions[time-in-range; value[time; 8:0:0]; value[time; 10:0:0]]

(2, 8): "Deny 10-12" and "Permit 8-10": target[];
  conditions[time-in-range; value[time; 10:0:0]; value[time; 10:0:0]]

(3, 8): "Deny 19-8" and "Permit 8-10": target[];
  conditions[time-in-range; value[time; 8:0:0]; value[time; 8:0:0]]

(5, 8): "Deny 20-15" and "Permit 8-10": target[];
  conditions[time-in-range; value[time; 8:0:0]; value[time; 10:0:0]]
```

```
(6, 8): "Deny 16-12" and "Permit 8-10": target[];
  conditions[time-in-range; value[time; 8:0:0]; value[time; 10:0:0]]

(7, 8): "Deny 22-20" and "Permit 8-10": target[];
  conditions[time-in-range; value[time; 8:0:0]; value[time; 10:0:0]]

(8, 9): "Permit 8-10" and "DenyAllOthers": target[];
  conditions[time-in-range; value[time; 8:0:0]; value[time; 10:0:0]]
```

Appendix G

Testing in Coq

This appendix contains the example found in Appendix E, but parsed by the java code into Coq.

```
Require Export RuleSet.

Definition rs : rule_set :=
(ruleCons permit (any :: nil) (any :: nil) (any :: nil)
  ((timeInRange (timeValue 32400) (timeValue 61200))::nil) ) ::
(ruleCons deny (any :: nil) (any :: nil) (any :: nil)
  ((timeInRange (timeValue 28800) (timeValue 43200))::nil) ) ::
(ruleCons deny (any :: nil) (any :: nil) (any :: nil)
  ((timeInRange (timeValue 36000) (timeValue 43200))::nil) ) ::
(ruleCons deny (any :: nil) (any :: nil) (any :: nil)
  ((timeInRange (timeValue 68400) (timeValue 28800))::nil) ) ::
(ruleCons deny (any :: nil) (any :: nil) (any :: nil)
  ((timeInRange (timeValue 54000) (timeValue 72000))::nil) ) ::
(ruleCons deny (any :: nil) (any :: nil) (any :: nil)
  ((timeInRange (timeValue 72000) (timeValue 54000))::nil) ) ::
(ruleCons deny (any :: nil) (any :: nil) (any :: nil)
  ((timeInRange (timeValue 57600) (timeValue 43200))::nil) ) ::
(ruleCons deny (any :: nil) (any :: nil) (any :: nil)
  ((timeInRange (timeValue 79200) (timeValue 72000))::nil) ) ::
(ruleCons permit (any :: nil) (any :: nil) (any :: nil)
  ((timeInRange (timeValue 28800) (timeValue 36000))::nil) ) ::
(ruleCons deny (any :: nil) (any :: nil) (any :: nil) (any :: nil)) ::
nil.
```

```
Open Scope nat_scope.
Eval compute in find_conflicts rs.
```

Running this code in Coq gives the following output:

```
 = (0, 1) :: (0, 2) :: (0, 4) :: (0, 5) :: (0, 6) :: (0, 7) ::
   (0, 9) :: (1, 8) :: (2, 8) :: (3, 8) :: (5, 8) :: (6, 8) ::
   (7, 8) :: (8, 9) :: nil : list (nat * nat)
```

This result, although not as descriptive as the last output has been formally verified to be correct.

This XACML to Coq parser gives two versions of Coq code. The first does not include the global target. This version is used to find conflicts within a single policy file. Since the global target is common between all rules, it is not necessary for finding conflicts. This version is only a direct parsing of XACML into Coq code. The second version includes the global target in each rule. Because of this, it can be used to find conflicts between two different policy files. This version although more complete, uses the unverified Java intersections to combine the global target with the rest of the rules. This is a limitation with the Coq code. In order for it to correspond with the firewall proofs, it was unable to return the intersections between rules, and so the Java algorithm had to be used. Future work includes updating the Coq code to include this functionality.

Bibliography

[1] Kenneth Appel and Wolfgang Haken. Every planar map is four colourable. *Bulletin of the American Mathematical Society 82*, pages 711–712, 1976.

[2] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorization Language. *Journal of Computer Security (JCS)*, 18(4):597–643, 2010.

[3] Venanzio Capretta, Bernard Stepien, Amy Felty, and Stan Matwin. Formal correctness of conflict detection for firewalls. *ACM Workshop on Formal Methods in Security Engineering*, pages 22–30, 2007.

[4] Adam Chlipala. *Certified Programming with Dependent Types*, 2010. http://adam.chlipala.net/cpdt/cpdt.pdf.

[5] Jan Chomicki, Jorge Lobo, and Shamim Naqvi. A logic programming approach to conflict resolution in policy management. *7th International Conference on Principles of Knowledge Representation and Reasoning*, 2000.

[6] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, pages 18–39, 2001.

[7] Joshua Finnis, Nalin Saigal, Adriana Iamnitchi, and Jay Ligatti. A location-based policy-specification language for mobile devices. *Pervasive and Mobile Computing Journal*, 2010.

[8] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, 2005.

[9] Georges Gonthier. Formal proofthe four-color theorem. *Notices of the American Mathematical Society 55*, pages 1382–1393, 2008.

[10] Florian Huonder. Conflict detection and resolution of XACML policies. Master's thesis, University of Applied Sciences Rapperswil, 2010.

[11] Sun Microsystems. Information about Sun's implementation of XACML can be found at http://sunxacml.sourceforge.net. Accessed in 2009.

[12] Tim Moses. *eXtensible Access Control Markup Language 3 (XACML) Version 2.0*, 2004.

[13] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[14] Prathima Rao, Dan Lin, Elisa Bertino, Ninghui Li, and Jorge Lobo. An algebra for fine-grained integration of XACML policies. *SACMAT 09: Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 63–72, 2008.

[15] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2009. http://coq.inria.fr/V8.2pl1/files/Reference-Manual.pdf.

[16] Vladimir Voevodsky. Univalent foundations project. http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/univalent_foundations_project.pdf.

[17] A.N. Whitehead. *Principia mathematica*. Number v. 1 in Principia Mathematica. Univ. Press, 1910.

[18] A.N. Whitehead and B. Russell. *Principia mathematica*. Number v. 2 in Principia Mathematica. University Press, 1912.

[19] A.N. Whitehead and B. Russell. *Principia mathematica*. Number v. 3 in Principia Mathematica. University Press, 1913.

[20] Ernst Zermelo. Investigations in the foundations of set theory. *From Frege to Gödel: a source book in mathematical logic, 1879-1931*, pages 199–215, 1967. Translated from Zermelo's original work, Untersuchungen ber die Grundlagen der Mengenlehre I, 1908.