# A Verified Algorithm for Detecting Conflicts in XACML Access Control Rules

Michel St-Martin[1] and Amy P. Felty[1,2]

[1] Department of Mathematics and Statistics, University of Ottawa, Canada
[2] School of Electrical Engineering and Computer Science, University of Ottawa, Canada
mstma016@uottawa.ca, afelty@eecs.uottawa.ca

**Abstract**

We describe the formalization of a correctness proof for a conflict detection algorithm for XACML (eXtensible Access Control Markup Language). XACML is a standardized declarative access control policy language that is increasingly used in industry. In practice it is common for rule sets to grow large, and contain unintended errors, often due to conflicting rules. A conflict occurs in a policy when one rule permits a request and another denies that same request. Such errors can lead to serious risks involving both allowing access to an unauthorized user as well as denying access to someone who needs it. Removing conflicts is thus an important aspect of debugging policies, and the use of a verified algorithm provides the highest assurance in a domain where security is important. In this paper, we extend our previous work on verification of conflict detection for Cisco firewall rules. We focus on several XACML constructs, the most complex of which are used for expressing time constraints. XACML is a significantly more expressive language than the one used to express firewall rules in Cisco products, and time constraints provide a good illustration of that expressive power. We propose an algorithm to find conflicts and then use the Coq Proof Assistant to prove the algorithm correct. We develop a library of tactics to help automate the proof.

## 1 Introduction

XACML (eXtensible Access Control Markup Language) [7] is a policy specification language that allows policies to be defined in a wide variety of domains. It is an OASIS [6] standard that is becoming more and more widely used, especially in recent years. Its expressive power allows access control policies to strike a balance between secure and flexible access; they must prevent access when there is a security risk and allow access when required, such as in a medical emergency when a doctor needs immediate access to medical records.

We present an algorithm for detecting conflicts in XACML, and implement it and prove its correctness in the Coq Proof Assistant [1, 3]. This paper presents work in progress. We do not cover all of XACML here, but do cover a significant sublanguage of XACML 2.0 [7]. The main restriction is that we do not cover all of the XACML's defined *functions*. This work extends the second author's work (with others) [2] on detecting conflicts in Cisco firewall rules, and reports on and extends the work in the first author's thesis [8]. Conflict detection is considerably more complex in XACML. The statement of correctness and its proof require considering a large number of cases, including many "corner cases" that were difficult to get right. In order to handle the added complexity, the work involved significant effort in automating proofs using Coq's Ltac facility [3]. This automation helped to both simplify the proofs and shorten the proof text.

In the next section (Section 2), we present the encoding of policies and requests in Coq. In Section 3, we present the conflict detection algorithm, and in Section 4, we briefly discuss its proof of correctness and the role of automation in this proof. We conclude and discuss related and future work in Section 5.

In the Coq code presented below, `Prop` is Coq's built-in type of propositions, while `bool` is in `Set` and is Coq's boolean datatype; the latter is used in the conflict detection program. We use Coq's built-in lists and integers. The `::` operator is an infix cons operator on lists, while `++` is infix append. We use both Coq's comparison operators on integers, such as `=`, `<`, `<=`, etc., and their boolean counterparts, whose names end in "`b`". For example, `(i<j)` is in `Prop`, while `(i <b j)` is in `bool`. The connectives `/\` `\/`, and `~` are used to construct propositions, while `&&`, `||`, and `negb` are the corresponding boolean operators. The files of the Coq formalization are available at `www.eecs.uottawa.ca/~afelty/arsec13/`.

## 2 XACML Policies and their Encoding in Coq

A rule is made up of a *target*, an optional *condition*, and an *effect* that indicates whether the rule will *permit* or *deny* access. A target groups together the *subject*, *resource*, and *action* components of a rule. Subjects, resources, and actions are restricted to a particular set of attributes, while conditions are more general. We present the subset of XACML we consider via an example policy encoded in Coq. Consider a policy for students to access computing and research laboratories at a university. We assume there are undergraduate and graduate computing laboratories where students work on their course work, and some number of research laboratories that each have graduate students assigned to them. We initially include the following rules: (1) anyone is allowed (undergraduates, graduate students, and professors) to enter the undergraduate computing laboratory during its opening hours, (2) graduate students and professors are allowed to enter the graduate computing laboratory during its opening hours, (3) the student with ID number 123 is allowed to enter the Formal Methods Research Lab during its opening hours, and (4) the student with ID number 456 is allowed to enter the AI Research Lab during its opening hours. We give the Coq definition of a rule first and then fill in the details.

```
Record rule: Set := ruleCons {eff: effect;  subjects: list srac;  resources: list srac;
                              actions: list srac;  conditions: list srac; }.
Inductive effect: Set := permit | deny.
```

A rule is a Coq record with 5 fields, whose names and types are specified. The first field has type `effect`, defined as an enumerated type with two elements via a Coq inductive definition. The remaining 4 fields contain lists of elements of type `srac`, which stands for "subject-resource-action-condition" since all of these elements use the same representation.

We present Coq code for rules (2) and (3) above before filling in the definitions of `srac` and other missing details.

```
Definition Rule2 := ruleCons permit
  (Grads++Professors) (GradLab::nil) (Enter::nil) (GlabHours::nil).
Definition Rule3 := ruleCons permit
  (G123::nil) (FMLab::nil) (Enter::nil) (FMlabHours::nil).
```

In the first rule, for example, the second argument to `ruleCons` is a list of subjects obtained by concatenating two lists, and the other elements of type `list srac` are all singleton lists. The subset of XACML functions we consider is defined by the following inductive definition:

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

The first function `any` represents an empty subject, resource, action, or condition, while the second is used to encode both XACML string equality and integer equality functions. (For simplicity, we encode general strings via integers.) For example, we have:

```
Inductive iValue: Set := intValue : Z -> iValue.
Definition G123:srac := (intEq (intValue 123)).       Definition G456:srac := ...
Definition Grads:list srac := (G123::G456::...::nil).
```

The type `iValue`, defined to represent integer attribute values, is used in rules and requests. Most of the XACML functions occurring in the above 2 rules are strings encoded as integers using `intEq`, including all the students (such as `G123` above) and professors, the labs, and the actions such as `Enter`. The only other XACML functions occurring in this example are those in the definitions of `GlabHours` and `FMlabHours`, which use `timeInRange` which appears as the last clause of the definition of `srac`. This clause and the two preceding it use the following encoding of time.

```
Definition MAX := 86400.     Definition Z' := {z:Z | 0 <= z < MAX}.
Inductive tValue: Set := timeValue : Z' -> tValue.

Lemma ThreeAm_rc : 0 <= 10800 < MAX.
Definition ThreeAm:tValue := timeValue (exist _ 10800 ThreeAm_rc).

Definition GlabHours:srac := timeInRange FourAm ThreeAm.
Definition FMlabHours:srac := timeInRange SixAm ElevenPm.
```

The type `Z'` encodes the subset of the integers that represent time, restricting time values to be between 0 and the number of seconds in 24 hours, and the special type `tValue` is used for time values appearing in rules and requests. Elements of this dependent type are a pair containing an integer and a proof that the integer is in the restricted range. The above example illustrates this representation for 3am. For any $z$ in the range $0 \leq z < MAX$, we write $z'$ to denote its corresponding value in `Z'`. For example, we write $10800'$ to denote `ThreeAM`. Other times are encoded similarly.

Note that in the time range `GlabHours`, the starting time is smaller than the ending time. The meaning is the range between 4am one day and 3am the next. Here, we are assuming that students can access the room anytime, except when it is closed for cleaning during the hour between 3am and 4am. This example provides a glimpse at the complexity of handling time ranges in policy rules. Determining if two rules conflict involves determining if there is any overlap in their time ranges, which involves considering both time ranges that fall within a single 24 hour period, and time ranges that "wrap around" midnight.

Suppose that student 123 has violated some university rule, and as a result is no longer allowed to enter research labs after 5pm. We add the following rule (and associated definitions).

```
Definition Violations := (G123::nil).     Definition After5PM:srac := timeGe FivePm.
Definition Rule5 := ruleCons deny Violations (any::nil) (Enter::nil) (After5PM::nil).
```

Note that this rule introduces some conflicts due to intersecting time ranges. Since the resource component of rule (5) is `any`, this rule applies to all labs. There is a conflict with rule (3), for instance, since student 123 is allowed and denied access to the FM lab during the hours between 5pm and 11pm. There is also a conflict with rule (2) from 5pm to midnight. Determining intersecting time ranges is, of course, a crucial component of our conflict detection algorithm.

A policy is a set of rules, represented here using Coq lists. In practice policies are used to evaluate requests. Here, we will restrict requests to contain 4 attribute/value pairs, one in each category: subject, resource, action, and condition. To illustrate, suppose student 456 wants to

```
Definition case1' (s1 e1 s2 e2: Z): bool := (s1 <=b e2 - MAX) && (e2 - MAX <=b e1).
Definition case2' (s1 e1 s2 e2: Z): bool := (s1 <=b e2) && (e2 <=b e1) && (s1 <=b s2).
Definition case3' (s1 e1 s2 e2: Z): bool := (s1 <=b s2) && (s2 <=b e1) && (e1 <=b e2).
Definition case4' (s1 e1 s2 e2: Z):  bool := e1 <=b e2 - MAX.

Definition allCases (s1 e1 s2 e2: Z)  : bool := (case1' s1 e1 s2 e2) ||
  (case2' s1 e1 s2 e2) || (case3' s1 e1 s2 e2) || (case4' s1 e1 s2 e2).

Definition e'2e : Z -> Z -> Z := fun s e' => if (s <=b e') then e' else e' + MAX.

Definition sracCheck: srac -> srac -> bool := fun sr1 sr2 =>
match sr1 with
| any => match sr2 with | timeLt (timeValue e2) => (0 <b e2)  | _ => true end
| intEq (intValue s1) => match sr2 with
     | intEq (intValue s2) => s1 =b s2  |  intGt (intValue s2) => s2 <b s1 ... end
| timeInRange (timeValue s1) (timeValue e1') => match sr2 with
     | timeInRange (timeValue s2) (timeValue e2') =>
       if (s1 <=b s2) then allCases s1 (e'2e s1 e1') s2 (e'2e s2 e2')
                      else allCases s2 (e'2e s2 e2') s1 (e'2e s1 e1') ... end
... end.
```

Figure 1: Detecting Overlap in Rule Components

enter the graduate lab at 3pm. The encoding in Coq of requests in general and this specific example is as follows.

```
Inductive reqValue: Set :=
| timeReq : tValue -> reqValue | intReq : iValue -> reqValue.
Record request: Set := requestCons {
 subj:reqValue;  res:reqValue;  acti:reqValue;  cond:reqValue }.

Definition G456R: reqValue := intReq (intValue 456).
Definition GradLabR: reqValue := intReq (intValue 2).
Definition EnterR: reqValue := intReq (intValue 0).
Definition ThreePmR: reqValue := timeReq ThreePm.
Definition Request1 := requestCons G456R GradLabR EnterR ThreePmR.
```

## 3 Detecting Conflicts in Policy Rules

Consider the conflict mentioned above between rules (3) and (5): student 123 is allowed and denied access to the FM lab during the hours between 5pm and 11pm. There is a conflict because all fields of both rules "overlap". The subject and action fields are exactly the same. The resource (the FM lab) in rule (3) is one of several labs covered by rule (5), and there is overlap in the two time ranges in the conditions of these rules (the first covers 6am-11pm, and the second covers 5pm to midnight). The key to the algorithm is correctly defining this overlap. Figure 1 defines the main function `sracCheck`, which detects overlap between 2 elements of type `srac`. This function must consider every case of each argument. Since `srac` has 6 constructors, there are many cases. The definition combines several of them, but still has 25 distinct cases. Only a small number are presented in the figure and explained below.

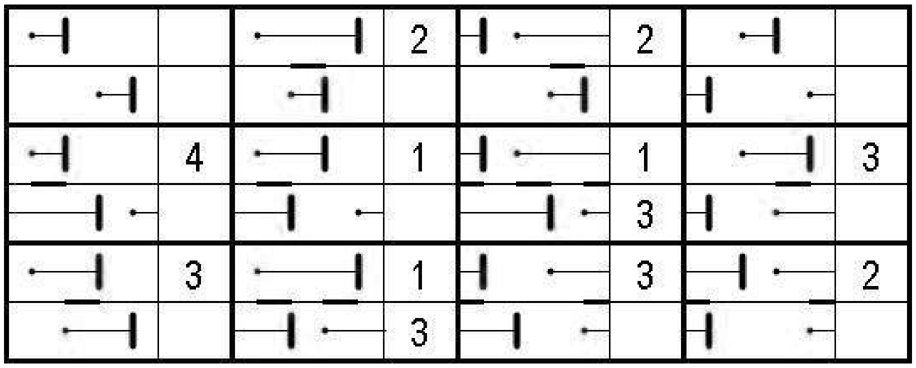

Figure 2: Illustration of Time Range Intersection Possibilities

The most complex case is when both arguments are time ranges, i.e., the first has the form `(timeInRange (timeValue s1) (timeValue e1'))` and the second has the form `(timeInRange (timeValue s2) (timeValue e2'))`. We will write these ranges as $[s_1, e'_1]$ and $[s_2, e'_2]$ for the sake of this discussion, where $s$ and $e'$ (possibly subscripted) denote the start and end points of a time range. We use the convention that whenever $[s, e']$ is a "wrap-around" time range (i.e., $s > e'$ such as in our example `GlabHours` in the previous section), we write $e$ to denote the value $e' + MAX$. This value is outside our allowed time values, but is used in situations where we want to force the end point of a time range to be greater than the start point. The function `e'2e` in Figure 1 performs this operation. In general, we say that a range $[s, e]$ is *normalized* when wrap-around time ranges have been adjusted in this way. Without loss of generality, we assume that $s_1 \leq s_2$ and illustrate all possible ways in which the two time ranges can intersect in Figure 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 $s_1$ and the bar represents $e'_1$. Similarly, the lower left box contains $s_2$ and $e'_2$ . 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. For example, in the first box, we have $s_1 \leq e'_1 < s_2 \leq e'_2$. The thicker bars between the ranges represent the intersection. In the right half of each part, we have numbers (or lack of one). These numbers represent how the ranges intersect, defined by the 4 Coq definitions at the top of Figure 1. Should there be a number in the right half of any part of the chart, then the corresponding case holds (for that particular ordering). Each case has a range in its definition which corresponds to the intersection between the two time ranges. In the four cases, we assume that the ranges are already normalized. For example, `case1'` holds if and only if we are in the three parts marked "1" in the chart. The distinguishing feature of these three boxes in the figure is that the range $[s_1, e'_2]$ is part (or all) of the intersection of the two time ranges. Note that in the definition of `sracCheck`, when both arguments `sr1` and `sr2` are time ranges, the function uses `allCases` to check whether or not any of the 4 cases hold, indicating that there is some intersection. We do not discuss all 12 possibilities in the figure and their correspondence to the Coq definitions of the 4 cases, but instead note that the soundness and completeness proofs verify that we have correctly covered all cases.

Notice that valid times include value 0, and thus it is possible to express the time constraint `(timeLt (timeValue` $0'$`))`. This is an empty constraint, and has no intersection with any time

```
listListCheck: list srac -> list srac -> bool.

Definition conflict_check: rule -> rule -> bool := fun rl1 rl2 =>
  match rl1 with (ruleCons e1 s1 r1 a1 c1) =>
  match rl2 with (ruleCons e2 s2 r2 a2 c2) => effectDiff e1 e2 &&
    listListCheck s1 s2 && listListCheck r1 r2 &&
    listListCheck a1 a2 && listListCheck c1 c2 end end.

find_conflicts: list rule -> list (nat*nat).
```

Figure 3: Details of Conflict Detection Program

range, which is the reason for the check `(0 <b e2)` in the case when `sr1` is `any` and `sr2` is `(timeLt (timeValue e2))`. This is a corner case we initially got wrong in our original proof. The value `any` intersects with all other time ranges and integer values. This test is also required in the case when `sr1` is a time range (not shown).

For the cases when `sr1` is `(intEq (intValue s1))`, first consider the simple case when `sr2` has the same form: `(intEq (intValue s2))`. There is overlap exactly when these two values are the same. In the case when `sr2` is `(intGt (intValue s2))`, then there is overlap with value `s1` if and only if `s1` is greater than `s2`.

An overview of the rest of the definition of conflict detection is found in Figure 3. The `conflict_check` function checks whether or not two rules intersect. In this function, `effectDiff` (whose definition is omitted) is called to see if one of `e1` and `e2` is `deny` and the other is `permit`. The function `listListCheck` is omitted (only the type is given). This function is called in `conflict_check` on the other components of the rule to check if each pair of lists overlaps. It does so by calling `sracCheck` in Figure 1 on each pair of type `srac`, one from the first list and one from the second. For example, the resource lists of two rules, `r1` and `r2`, overlap if at least one of the resources from `r1` overlaps with one from `r2`.

This algorithm handles the full generality of XACML subjects, resources, and actions, but not conditions. In XACML, for a rule to apply, all conditions must apply, which means that in order for two rules with conditions to conflict, there must be overlap in all conditions. Our algorithm correctly handles the restriction of XACML to just one condition in each rule. Extending our algorithm to handle the full expressive power of conditions is the subject of current work.

To detect if two rules conflict, we go through the policy, represented as a list of rules, to find all conflicts. This part of the implementation comes directly from [2], and we do not repeat it here. Figure 3 shows the type of the main function called `find_conflicts` (which uses two other helper functions not shown). Together, they define a recursive traversal of the list of rules. For each rule $r$ in the list, all rules occurring after $r$ are tested for conflict with $r$ by calling `conflict_check`. The function returns a list of pairs of indices corresponding to those rules that conflict.

## 4 Correctness and Proof Automation

The following lemma states that if the `conflict_check` program returns true for two input rules, then these rules do indeed conflict, according to the definition of `rule_conflict`.

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

We omit the definition of `rule_conflict` (we refer the reader to the associated proof scripts), but just note that it directly expresses what it means for two rules to conflict, as described earlier: there exists a request such that one rule permits this request and another denies it. This lemma states that if the `conflict_check` program returns `true` for two input rules, then these rules do indeed conflict. Conversely, completeness (not shown) states that if two rules conflict according to the definition, then given these 2 rules as input, the program will return `true`.

Most of the work and lines of code in the Coq formalization come from the proofs of these soundness and completeness lemmas. For example, we proved a soundness and completeness lemma for each pair of XACML functions that we cover. As an example, the following lemma expresses soundness for the case when both elements of the pair are `timeInRange` functions.

```
Lemma rangeSound: forall (s1 e1' s2 e2':tValue) (sr1 sr2: srac),
  sr1 = timeInRange s1 e1' -> sr2 = timeInRange s2 e2' -> sracCheck sr1 sr2 = true ->
  exists r:reqValue, (valueMatch r sr1) /\ (valueMatch r sr2).
```

This lemma states if the two time ranges overlap (function `sracCheck` returns true), then there is indeed a request whose time falls within both rules' time constraints, as defined by `valueMatch`. The definition of `valueMatch`, also not shown, is similar to `sracCheck` except that it compares a rule component to a component of a request. (Since a request is simpler than a rule, there are slightly fewer cases.)

The automation built into the custom tactics that we developed is especially important for proving such lemmas. The tactics were specifically designed to reason by cases, and to do so uniformly across all the cases that appear in the numerous lemmas needed for soundness and completeness. For example, `sracCheck` in the above lemma unfolds to 25 cases. All except the case when both arguments are time ranges are quickly ruled out, but then this particular case expands to 8 cases because it involves an `if` statement where each branch contains `allCases`. As a result of defining these tactics, although applying them requires a significant amount of computation, the proofs remain relatively short.

Soundness and completeness of the `find_conflicts` program is obtained fairly directly by extending the definition of of `rule_conflict` to express the notion of two rules at particular indices within a policy (represented as a list of rules) being in conflict, and is proved fairly directly from the lemmas showing soundness and completeness of each XACML function (such as `rangeSound` described above).

## 5 Conclusion

We have presented an algorithm for detecting conflicts in XACML rules for a subset of the XACML language that includes fairly complex conditions such as time constraints, and we have verified its correctness in Coq, along the way defining tactics to automate proofs, designed to be general enough to be reused in many of the lemmas in the proof development.

There is much work on developing algorithms and tools for analyzing policies. We mention only other work that focuses on conflict detection and applies to XACML. With regard to our own past work, as mentioned, this work extends our work on conflict detection for firewalls [2]. We have also been involved in work on a tool for administering XACML policies, with a focus on usability for policy administrators that do not necessarily have a technical background [9, 10]. An implementation of conflict detection (unverified) was also part of that work.

Huonder [4] developed a conflict algorithm that is generic in the sense that in order to work with concrete XACML policies it must first be extended with definitions (of XACML functions and intersections). He also proposes ways to resolve conflicts. For example, he proposes automatically repairing them by replacing the rule set with an equivalent one without conflicts. In order to do so, whenever there is a conflict, it has to be resolved according to the default resolution policy. For example, if "first-applicable" is chosen, and a rule that denies the request comes first in the policy, then all conflicting rules that appear later have to be changed so that they don't cover this request. In other words, the overlap has to be determined and removed. His algorithms are quite different from ours and not verified. Also, this may resolve the conflicts, but does not necessarily remove the bugs, and furthermore an automatically modified policy makes it harder for the policy administrator to read and understand it. In contrast, our approach provides an opportunity for the administrator to examine each conflict and determine the best way to resolve it him/herself.

In [5], a formal tool is used to analyze policies. Policies and requests are modeled in the Alloy analyzer, and first-order queries are presented to answer questions such as whether or not there are two rules in a given policy that conflict. The subset of XACML considered is simpler than ours. For example, conditions are not considered, and thus complications such as those resulting from time constraints are not handled.

We have discussed the main direction of future work in the previous sections, namely, extending the results to cover the full expressive power of XACML. The biggest challenge is to cover all functions and logical operators allowed in conditions, which is our current focus.

## References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

[2] Venanzio Capretta, Bernard Stepien, Amy Felty, and Stan Matwin. Formal correctness of conflict detection for firewalls. In *ACM Workshop on Formal Methods in Security Engineering*, pages 22–30. ACM Press, 2007.

[3] Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2009. `coq.inria.fr/distrib/V8.4/refman/`.

[4] Florian Huonder. Conflict detection and resolution of XACML policies. Master's thesis, University of Applied Sciences Rapperswil, 2010.

[5] Mahdi Mankai and Luigi Logrippo. Access control policies: Modeling and validation. In *5th NOTERE Conference*, pages 85–91, 2005.

[6] OASIS. eXtensible Access Control Markup Language (XACML) TC. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`, 2004.

[7] OASIS. *XACML Version 2.0*, 2004. `docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`.

[8] Michel St-Martin. A verified algorithm for detecting conflicts in XACML access control rules. Master's thesis, University of Ottawa, 2011.

[9] Bernard Stepien, Amy Felty, and Stan Matwin. A non-technical user-oriented display notation for XACML conditions. In *E-Technologies: Innovation in an Open World, 4th International MCETECH Conference*, pages 53–64. Springer LNBIP, 2009.

[10] Bernard Stepien, Stan Matwin, and Amy Felty. Strategies for reducing risks of inconsistencies in access control policies. In *5th International Conference on Availability, Reliability, and Security*, pages 140–147. IEEE Computer Society, 2010.