Automatically Summarizing all the Problems of a Rule-Based System

Jean-Claude Royer, Massimo Tisi

Automatically Summarizing all the Problems of a Rule-Based System

Jean-Claude Royer and Massimo Tisi

IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France
{jean-claude.royer, massimo.tisi}@imt-atlantique.fr

Abstract. Looking for conflicts in software policies is a critical activity in many contexts, like security, expert systems, and databases. These policies are often written or specified as logical rule-based systems, which is a flexible and modular formalism. However, this formalism does not guarantee by itself the safety of the system. While several works address conflict detection, as far as we know there is little research on making explicit all these problems in an abstract and simplified form. In this paper, we compare different techniques to extract all the problems in a rule-based system. We define an exact algorithm and an application on three middle-size case studies, but as expected with poor time performances. We improve this with a combine algorithm and a few simple heuristics that accelerate the computation of all the problems in the case studies by several orders of magnitude.

keywords: Rule-based systems, Rule conflicts, Satisfiability

1 Introduction

While rule-based systems are popular due to their flexibility and modularity, techniques and tools are still needed for assisting users in their practical management. One of the most critical aspects is detecting (and correcting) conflicting situations. There exist already several approaches, specifically in the area of security policies, to check for conflicts [11,12,1,4], and even to fix them [13,9,5]. However, these approaches are restricted to specific languages (e.g. for firewall policies), or only consider a limited set of problems or need a rule ordering. Detecting problems in a context without priority, with quantifiers and unrestricted predicates and functions is still a complex manual activity. We argue that an automated approach for making explicit all these conflicts in an understandable form would be of great help during the problem-fixing phase. Users could use this problem summary to effectively select the most critical problems, that may be addressed first. For example, several conflicting instances may be coming from the same problem and of course we should prioritize removing the general problem rather than each instance one by one. We already defined a solution to retrieve the list of all conflicts in [4] but in realistic cases this raw list is too long and its simplification is difficult and expensive. Another point is that we

should cope with a possible input sanitization. For instance, logically in an access control policy DENY and ALLOW is a conflicting situation. However, it is not a real problem since such a request will not be submitted to the engine running the rule system.

Obtaining a simplified view of all the conflicts is computationally expensive for several reasons: requests enumeration, use of variables, number and logic of the rules, simplification of the found problems. An additional complexity comes from the fact that the processing intertwines syntactic and semantics aspects. While the theoretical complexity is high, we look for algorithms and heuristics which can succeed in processing middle-size case studies in a reasonable amount of time. We provide the following contributions:

- We discuss a first simple but inefficient algorithm and various other ways of enumerating these problems.
- We describe an exact solution that improves time performance and size of the result. For instance we can solve a use case with 61 rules in nearly 50 minutes resulting in 6 problems. However, this exact solution is too slow for more complex case studies.
- We propose a heuristic that allows to solve our three use cases in less than 60 s and finding the same problems. This heuristic is based on the idea of exposing the most abstract problems first.

The content of this paper is structured as follows. Section 2 presents a motivating example and provides the necessary background and definitions to understand our approach. Section 3 describes related work in the area of rule-based systems and checking for conflicting rules. Section 4 discusses several options to compute the problems and gives additional information about our processing algorithms. In Section 5 we present a first algorithm and its evaluation on our case studies. The main contribution is our `combine` algorithm which is described in Section 6 and evaluated. Lastly, in Section 7 we conclude and sketch future work.

2 Motivation

In Figure 1.1 we have three rules of a rule-based system in predicate logic. We will follow a syntax inspired by the Z3 solver [17], but implication is denoted by `=>`.

Listing 1.1. A simple example

```
And(allow(X), deny(X)) => False              #1
student(X) =>  allow(X)                      #2
And(student(X), repeating(X)) => deny(X)     #3
```

We know that the language of system inputs (or *requests*) allows for any logical combination of the predicates `student` and `repeating`. The language of system outputs (or *replies*) is compound from the vocabulary `allow` and `deny`. Note that the conclusion of the first rule is false. The rule is not describing an expected

reply but defines relations between the conclusions `allow` and `deny`. We call *explicit unsafe* the rules whose conclusion is false.

Our aim is to define a tool which is able to summarize all the correct but undefined requests. A correct request will be one compliant with the input grammar. An undefined request is responsible for contradictory replies, in other words its conjunction with the rule system makes it unsatisfiable. A correct and undefined request will be called a *real problem* or a problem for short. For instance, we expect our tool to return the result shown in Listing 1.2 when applied to the previous example.

Listing 1.2. Undefined requests and problems in the example

```
---------- undefined requests --------------
 And(allow(X), deny(X))
---------- real problems --------------
 And(student(X), repeating(X))
```

The difference between undefined requests and real problems is important because the first kind is not expected to be submitted to the rule system. Real problems are called conflicts, inconsistencies, etc. in literature, and can generally lead to various kinds of failures.

Since we have a grammar for the requests we expect to be able to decide if any satisfiable logical expression represents a request. We do not consider a grammar for the replies, since it is not useful in our task of summarizing the problematic requests. We may consider such a grammar in future work, for analyzing the correctness of the responses.

Our previous work at [4] is able to generate a list of all the undefined requests of a given rule-based system. When applied to a full real-world system, this list becomes easily unmanageable both because of the number of undefined requests and the size of each one of them. Moreover our previous work cannot distinguish between undefined requests and real problems.

Thus our objective is to present the user with a short list of simpler problems, by 1) performing automatic simplification and 2) considering the requests language throughout the analysis. This objective has in general a higher computational complexity. Thus we need to improve previous work in several ways: more complex preprocessing, simplification, use of hash coding and binary representations.

2.1 Case studies

Our initial objective was to define an algorithm to process three middle-size examples in a reasonable amount of time. Our time limit to compute a simplified description of all the problems in the case studies was 10 hours. The first example (RBSSIM) is an expert system for recommending investments, the second (ContinueA) a classic access control policy for a conference management, and the third (Healthcare) an administrative role-based access control for healthcare management. Note that these three examples have free universally quantified variables, but in addition the healthcare one has few rules with universal quantifiers inside its conditions.

Description	Rules	Kind of logic
RBSSIM		
expert system	36	unary predicates and arithmetic (`Int.<, >=, *`)
`https://courses.csail.mit.edu/6.871/Assignment2/RBSSim.pdf`		
ContinueA		
XACML policy	47	binary and unary predicates
`http://cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/`		
Healthcare		
ARBAC healthcare	61	1,2,3-nary predicates and `Int.<`
`http://www3.cs.stonybrook.edu/~stoller/ccs2007/`		

Table 1. Our three case studies

2.2 Background and notations

In our tool and our use cases we use FOL and a logical solver (Z3) but the technique can be applied to a large set of logical frameworks. Of course the method depends on the ability of the solver to manage such a logic.

A *rule* is a logical implication, taking the form of $D \Rightarrow C$, with D being the condition and C the conclusion of the rule, expressed in a logical language possibly with free variables noted by $*$. Condition and conclusion are any kind of FOL expressions, there is no restriction here. A *rule system* $(\forall * R)$ is simply a satisfiable conjunction of rules with possibly free universally quantified variables $(\forall * \bigwedge_{1 \leq i \leq n}(D_i \Rightarrow C_i))$. *Requests* are FOL expressions. When they are sent to a rule system at runtime, they will be evaluated against all rules in that system to generate *replies* (which are also FOL expressions). A request is called *undefined*, if it is satisfiable by itself, but unsatisfiable when in conjunction with R. Depending on the implementation, R would give contradictory/unsatisfiable replies to the undefined requests, therefore making the system unrealizable.

A rule system will be provided with an input grammar REQ defining the language of requests. We note $D : REQ$ meaning that D is compatible with the grammar REQ and $D ! REQ$ if it is not.

Let A_i be a set of logical atoms, that is propositional variables P_j or predicate calls $pred_k(v_1, ..., v_n)$. A *conjunction* will be any AND combination of such atoms or their negation.

3 Related work

There is a large amount of work aiming at detecting conflicting situations in rule-based and related systems. The survey in [25] observes the lack of formal verification methods for this task. Indeed many existing validation and verification approaches for these systems are based on testing. Validation and verification techniques for expert systems and database management ([6]), for Web policies and contracts ([20]) or for knowledge based systems ([18]) already exist. Our domain is rather security policies where the problem of conflicts has

been intensively studied. In surveys on security [11,12,1], conflict detection is a central problem but it is typically treated together with other tasks like finding bugs, redundancies, misconfigurations, etc. There exist already a few efficient algorithms to statically detect conflicts but often they are dedicated to specific languages, and the goal is not to find all the problems in a simplified form. Amongst them [10,8,14,19,3,2], a more exhaustive list is available in the survey [22]. There are also several papers about finding and fixing such problems, we refer to [16] for a bibliography. Researchers try to find automatic fixes for various kinds of bugs, e.g. redundancies [13], misconfigurations [9]. Son et al. [23] repair access-control policies in Web applications. Wu focuses on detecting inconsistency bugs among invariant rules enforced on UML models [24].

Work on extracting *minimal unsat core* and *maximal sat core* (e.g. [15,21]) are different from ours. Here we consider the minimality of a satisfiable expression but which is unsatisfiable in conjunction with another expression. The MUS computation needs the knowledge of the requests and its result is useless to compute the minimal sum of the requests. Our previous work [4] deals with finding all the undefined requests in a rule system. In [5] we also study automatic repair in this context. Our current work has a more ambitious goal which is finding a simplified description of all the undefined requests compliant with an input grammar. This objective is more complex and computationally more expensive, thus the algorithms we will describe in the following sections are very different from previous solutions.

4 Solving the problem

In this section we discuss some key issues we need to address, and point out some possible solutions. We are looking for all problems, i.e. all requests that are *correct* (w.r.t. the request language) and *undefined* (w.r.t. the rule system):

- In the general case there are $3^{\#A} - 1$ correct requests, with $\#A$ the number of atoms in the request language. In our first example there are only eight correct input requests, that are compound of the predicates `student` and `repeating`.
- By using the method in [4] we can efficiently enumerate all the undefined requests (including requests that are not correct w.r.t. the request language) by considering a number of rule combinations that is exponential w.r.t. the number of rules. In the example, the method in [4] would compute $2^3 - 1$ rule combinations leading to a set of non simplified undefined requests.

Finally, to list all problems we must intersect these two sets. This problem could be solved with a logic description of the correct requests, however, it is not always possible to do that. Finally we need to use the solver to check for satisfiability or unsatisfiability which is an expensive task in general. Thus performing this task is expensive for three main reasons: the number of correct requests, the number of undefined requests and the use of a solver. Our final goal is identifying the smallest set of minimal (simpler) problems. Thus we experiment and compare several solutions.

4.1 Input grammar

A first question is how to represent the grammar for requests and how to check that an expression is compliant with it. The number of such expressions is often infinite but we need of course a finite way to check compliance. This is a syntactic problem on the structure of the input. We can try to represent the grammar with a logical formula, but this is in general not convenient. For instance, we cannot logically enable requests containing either `P` or `Not(P)`. Normal forms are often used to solve this kind of issue, that requires checking both syntactic (here input compliance) and semantic (here undefinedness) properties.

Intuitively, a request is a conjunction of some atoms (positive or negative). The reason is that a request should match one or more rule conditions and using disjunctive normal forms for conditions we can define an equivalent rule system with conjunctions of atoms as conditions. Thus our hypothesis here is to consider that any request is a conjunction of predefined atoms which are selected by the specifier and which define the input grammar. In our prototype we choose to consider expressions at the top-level, that is under a Boolean operator or starting with a quantifier. This is a sensible and simple choice but different ones are possible (with consequences on the number of atoms and the computation time). In order to simplify and optimize the processing we renamed the rules according to the chosen expressions. We rename these sub expressions (Tseytin transformation) and store new predicate definitions in the solver. Note that our rule system can contain quantifiers, functions in conditions and conclusions.

We consider that the input grammar is represented by a set of expressions. In our simple example we have `REQ = {student(X), repeating(X)}`, where `X` is a free universally quantified variable. Each expression in `REQ` has an associated atom with this expression as definition. Thus applying renaming we will get an equivalent system with a set of definitions and rules rewritten with atoms.

Listing 1.3. Renaming

```
 ---------- definitions --------------
OrderedDict([(P_1(X), allow(X)), (P_2(X), deny(X)),
             (P_3(X), student(X)), (P_4(X), repeating(X))])
 ---------- rules renamed --------------
<And(P_1(X), P_2(X)) => False>
<P_3(X) => P_1(X)>
<And(P_3(X), P_4(X)) => P_2(X)>
```

The cost of this renaming step is linear and we use Z3 hashcode to check expression equality. Our hypothesis on the input grammar could seem a bit strict, nevertheless it is rather flexible and it is possible to consider cases where an atom is legal but its negation is not. However, in the following and in our experiments we strictly embrace this assumption, which will be discussed in detail later.

4.2 Identifying the real problems

The above hypothesis greatly simplifies the automatic distinction between correct and incorrect requests. The major problem is to identify the requests which are problems and to get a simplified set of these problems. And of course this

may be time consuming due to the number of requests to test, the use of a logical solver, and the time to simplify expressions. Many different ways are possible, we give here some hints about the ones we have experimented and we will detail a naive solution and a more efficient one in the next sections.

Our first approach was to reuse our previous work in [4]. The basic idea is to compute all the undefined expressions as defined by the following formula $\bigvee_{I \in 2^n} \exists * \ (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j \neg \bigwedge_{i \in I_1} C_i)$, where I_1 (respectively I_0) is the set of positive (respectively negative) rules in the binary combination of the rules. This computation was done in [4], using an iterative and a sorting method. The first problem is that the method succeeds in less than 2 minutes with the ContinueA example but takes more than 2 hours for the two other cases. Furthermore, the undefined information is numerous and complex because of lack of simplifications. Even by using a normal form for the rules, the computing process was too expensive. This way was unsuccessful also because some problems result from the conjunction of undefined requests as we will see later.

The most naive approach is testing: generating all the correct requests and identifying the ones which in conjunction with the system are unsatisfiable. The problem is that the set of requests can be infinite, and in many practical cases it is finite but too large. Furthermore, if we have free variables in our rule system we should check for all free variable instantiations, since a real problem has an existential nature. But freezing free quantified variables and using normalized requests leads to a possible solution which is called `enumerate` and will be described in the next section. We present it because this may be the simplest solution which provides partial results.

Another approach could use the fact that any real problem is included in the negation of the rule system ($\neg \forall * R$) which is equivalent to $\bigvee_{1 \leq i \leq n} \exists * (D_i \wedge \neg C_i)$. But one real problem could be included in several of the terms of this union, take for instance to `A => C` and `B => Not(C)`, with `REQ = {A, B}` which combined produce a problem `And(A, B)`. Thus we should compute all the exponential number of combinations between the rules, and we still have the problem of simplification.

In both the two last cases there is an exploration of a set of requests. Since we expect to get minimal (in the abstract sense) expressions, a breadth-first search is preferable. We could also note that once a node is undefined all descendants are undefined which limits the search process. Conversely, if one node is known to be defined, then all its ancestors are also defined.

4.3 Problem simplification

We consider the classic two-level minimization principles of Boolean logic [7]. However, because of the cost of this task we assume a *prime* algorithm which computes a list of prime implicants covering all the problems. We generally do not manage directly MIN-terms (that is a maximal conjunction of atoms) but rather a binary list representing a sum of product terms. We implement our proper solution for the prime algorithm but exact or more efficient solutions, like the Quine McCluskey, Expresso or SCHERZO algorithms, can be used. We

do not go in these technical details here because, as we will see, the simplification solution is not essential to the algorithms and our final result.

4.4 Preprocessing

This subsection summarizes some preliminary steps before discussing our two algorithms. Our algorithms exploit some initial transformations of the rule system.

Simplification. The rule system is checked for satisfiability, then redundant rules are discarded. Of course this step depends from the rule ordering but this is not an issue.

Renaming. As discussed above, we apply a renaming step which allows to simplify the rule system. The user chooses a set of expressions occurring in the rule system and this step computes an equivalent system using the principle of Tseytin renaming. We suggest to use expressions close to the top-level of conditions and conclusions, otherwise the parsing and the further computation will be too expensive. Note that we do not assume that the definitions used for renaming are logically independent.

Binary conversion. A conjunction of atoms can be converted into a binary list of digits. The reference ordering is obtained from the sequence of definitions (ordering matters here) and 1 stand for positive, 0, for negative and -1 for a don't care bit. Thus we defined a type `Binary` as `List[{0, 1, -1}]` which is used by some of our algorithms.

Negation of the rule system. To compute the negation of our rule systems we reuse tactics from Z3. We define a compound *tactic*:
`Then(nnf, simplify, Repeat(OrElse(split-clause, skip)), simplify)` to compute the sum of product terms of the negation of the renamed rule system.

5 The enumerating process

The simplest way to identify the problems is to enumerate the MIN-terms representing input requests and to check them against the rule system using the solver. We note $\#REQ$ the number of distinct atoms (or definitions) used in the renaming process and defining the input grammar. This is expensive since we have $3^{\#REQ} - 1$ correct requests, furthermore we do not get a simplified result. Since we need to extract simpler problems first, we enumerate the requests starting from single atoms and their negation. Generating these requests in a breadth-first order has higher chances to obtain a simplified result. However, this is not sufficient and we get a long list of problems. It is possible to use our `prime` algorithm, but it needs a binary conversion and this consumes more time. We also remark that checking for unsatisfiable requests before their undefinedness is time expensive, since it calls the solver for each request. Indeed, there is no undefined request involving the first 30 rules of the ContinueA and the Healthcare examples, but there are more than six thousands undefined requests on the first 20 rules of the RBSSIM case.

The `enumerate` algorithm is rather straightforward from a breadth-first search of the atoms combinations. It starts with each atom and its negation then computes all the combinations from the smallest to the largest.

Listing 1.4. The Enumerate algorithm

```
enumerate_check(problems, lprop)
  IF lprop THEN
  current = lprop[0][0]
  pb = check_undefined_request(current)
  pbnot = check_undefined_request(Not(current))
  IF pb THEN
     problems.append(current)
     IF pbnot THEN
        problems.append(notcurrent)
        enumerate_check(lprop[1:])
     ELSE
         # notcurrent is defined
         res = check_conj(notcurrent, lprop[0][1])
         IF len(lprop[1]) > 1 THEN
            lprop[1] = [lprop[1][0], res]
            enumerate_check(lprop[1:]) #
         # ---
     # ... other cases are similar
  RETURN problems
END # --- enumerate_check
```

The sketched algorithm in Listing 1.4 starts from the single atoms in REQ. The conjunctions are represented by list of atoms in REQ. `lprop` is used to store visited nodes. It is a list of lists, each one with the current node to visit and a list of successors. It is initialized with a list of an atom and an empty list for each atom in REQ. The algorithm checks each atom and its negation then if a problem is found it is stored. The search continues on the rest of the list. When the atom is defined a conjunction is built and checked with `check_conj`. This returns a list of new conjunctions to visit which is stored at the end of the list for breadth-first processing. The `check_conj` function builds a new conjunction from `prop` and each conjunction in `listoflconj`. Then it checks for undefinedness and stores the problems. This function returns all the remaining defined conjunctions to visit in the next step. It uses a `check_undefined_request` function to check a conjunction of atoms for undefinedness against the rule system. If the solver gives an unknown reply, a message is printed to alert the user. There are three ways to tackle this case. The user can change the solver timeout to remove this uncertainty but he may also fall into an undecidable case. She may change the specification or use a more suitable solver for that logic. Finally, if she continues, she will not be sure to get all the problems.

Name	*#rules*	*time* (s)	*#problems*	*#checking*
RBSSIM	20	538	6736	65730
Healthcare	30	1563	26	106578
ContinueA	30	36284	1071	3672198

Table 2. Few results with the `enumerate` algorithm

With the ContinueA policy and the first 30 rules the enumeration takes 10 hours. Our experiments (see Table 2) show that after nearly 20 or 30 rules it is tedious to wait for a long list of complex problems. These experiments have been performed on a Mac Book Pro under High Sierra, 2.9 GhZ, 16Go of RAM, pydev 7.2.1, Python 3.7.2 and the z3-solver package 4.8.5.0. When it is sensible, the shown time results from an average of 10 runs. The full source code and examples can be found at the tool website[1]. In the table, *#checking* is the number of solver calls, which is an indicator of the real complexity of the task.

6 The combining process

We have experimented a set of algorithms based on the principle that undefined requests are included in the negation of the rule system. Thus we compute a sum of product terms with the Z3 tactic for the negation of the rule system, and we get $\bigvee_{1\leq i\leq m} U_i$ a union of conjunctions of atoms. Each U_i can be viewed as a conjunction $R_i \wedge N_i$ where $R_i : REQ$ and $N_i ! REQ$. Furthermore, we can assume that both expressions are not false. We also assume that R_i is not true, since we have no problem included in N_i. To compute all the problems we use the following principle.

Property 1 (Combining principle).
Let $\vee_{1\leq i\leq m}(R_i \wedge N_i)$ with R_i neither false nor true and N_i not false.

- If $\vee_{j\in J\subseteq\{1\leq i\leq m\}} N_j$ is true $\wedge_{j\in J} R_j$ is satisfiable then $\wedge_{j\in J} R_j$ is a problem.
- If a problem pb is included in the union $\vee_{1\leq i\leq m}(R_i \wedge N_i)$ then it exists $J \subseteq \{1 \leq i \leq m\}$ such that $\vee_{j\in J\subseteq\{1\leq i\leq m\}} N_j$ is true.

The sufficient condition for a problem is simple to see. The necessary condition can be explained as follows. If pb is a problem included in the union then $pb = \vee_{1\leq i\leq m}\ pb \wedge R_i \wedge N_i$. Eliminating the unsatisfiable case of the union we get $pb = \vee_{1\leq j\leq k}\ R'_i \wedge N_i$ with the same hypothesis as above. Then we decompose the equation with all R'_i combinations, and we get a disjoint union with one term $\wedge_{1\leq i\leq k}\ R_i \wedge (\vee_{1\leq i\leq k} N_j)$. Since we have a disjoint union of $!REQ$ normal forms then the problem is a disjoint union of problems included in each term. There is no problem in the terms except in $\wedge_{1\leq i\leq k}\ R'_i \wedge (\vee_{1\leq i\leq k}\ N_j)$. It shows that the above condition is necessary. Concretely we can use this result as follows: compute all the conjunctions of the REQ parts in $\neg R$ and check them for undefinedness.

We use the simple *tactic* from Z3 to reduce the negation of the rule system to a sum of conjunction of atoms. This tactic was chosen because it is efficient but more sophisticated algorithms are possible. We get a list of atom conjunctions which is converted in a list of binary representations, called `binary:Binary` to be short. From this list of `Binary` we can first extract some problems. We also simplify the list for convenience, but these steps are not essential. A second step is to reduce each binary in the list, since real problems should be compliant

[1] `https://github.com/atlanmod/ACP-all` branch `fixall_june2019`

with the input grammar. We get a list of *reductions* (`reduction`:REQ), that is binary reduced to atoms occurring in the input grammar. Then several ways are possible to compute all the real problems: i) To expand each reduction and to check it for undefinedness, and ii) To compute all the conjunctions and to check them for undefinedness. There are also several possible optimizations: to discard cases when the reduction is unsatisfiable, to check if a conjunction expression has been already seen, or to check if it is a MIN-term (that is without don't care bit) since by combination either we get an unsatisfiable request or the same expression. Experiments have shown that these techniques outperform the previous enumerative method. Table 3 allows comparison with the previous Table 2 on the same first rules of the three case studies. The table shows that times are obviously improved. This is also true for the size of the set of problems, even if we deactivate the simplification algorithm. Logical equivalence of the found problems was also checked using the Z3 solver but only for the 20 first rules.

Name	*#rules*	*time* (s)	*#problems*	*#checking*
RBSSIM	20	45	4	10357
Healthcare	30	26	5	3036
ContinueA	30	243	6	32151

Table 3. To compare `combine` and `enumerate` algorithms

However, this combine method is still rather expensive due to the huge number of expressions to check for undefinedness. The best configuration of optimizations is described in the following section.

6.1 A complete solution

In Listing 1.5 the initial steps are used to compute a sum of product for the negation of the rule system. Thus `binaries` contains all the undefined requests. This list is then filtered to extract initial problems (that are compliant with `REQ`) and simplified. Then a list of reductions is extracted and cleaned for double or True cases (that is `[-1*]`). The loops are computing all the distinct combinations of reductions by stage of same number of combinations. Each time a new combination (`common`) is computed from existing combinations and reductions. If the combination is not trivially unsatisfiable or not already included in the set of problems, it is checked for undefinedness. The `nextcombined` list memorizes the defined combinations with an index (`nextlasts`) of the last `allreq` element in order to only include distinct combinations. Finally, the set of problems is checked for unsatisfiable cases and simplified.

This process is optimized using few additional elements which have a real effect on performances. The implemented algorithm manages a list of already seen expressions to avoid double checks. In case `common` is not a problem and if it has no don't care bit or `J == size-1` it is not necessary to combine it further.

Listing 1.5. The Combine algorithm

```
1  binaries = tactic(Not(R))
2  filtered, problems = remove-problems(binaries)
3  allreq = reduction(filtered)
4  combined = allreq
5  lasts = [0 .. size(allreq)-1]
6  WHILE size(combined) > 1 DO
7       nextcombined = []
8       nextlasts = []
9       I = 0
10      WHILE (I < size(combined)) DO
11           current = combined[I]
12           last = lasts[I]
13           FOR J in [last .. size(allreq)-1] DO
14                other = allreq[J]
15                common = compute-and(current, other)
16                IF (common != []) THEN
17                   IF (common not included in problems) THEN
18                      IF (check-undefined(R, common)) THEN
19                         add(problems, common)
20                      ELSE
21                         add(nextcombined, common)
22                         add(newtlasts, J)
23           ENDFOR J
24       ENDWHILE I
25       combined = nextcombined
26       lasts = nextlasts
27 ENDWHILE combined
28 RETURN prime(remove-unsat(problems))
```

Note that the test `common != []` is only a sufficient syntactic check for unsatisfiability. Hence a final check for unsatisfiability is done to remove unsatisfiable requests from the list of real problems (there were one in the Healthcare and four in the RBSSIM example). If it is performed during the exploration, it is much more costly than doing it only at the end, like with the `enumerate` algorithm. The results are shown in Table 4. It takes nearly 50 minutes to entirely process

Name	*time* (s)	*#problems*	*#checking*	*#level*	*#without*
RBSSIM	378039	13	3252166	19	13
Healthcare	2892	6	50107	14	6
ContinueA	253108	64	1548997	18	111

Table 4. Results with the `combine` algorithm

the Healthcare case, but much more for the two other examples. The number of simplified problems for the two first cases are low. There is a large number of problems with ContinueA, it is not surprising since the system results from removing the combining rules in the XACML original example. Indeed without final simplification we get the number of problems in the *#without* column. This shows that a simplification algorithm is not essential but of course useful if it is not too much time consuming. *#level* is the number of combination stages, meaning that at most *#level* stages are needed to compute all the possible combinations of the reductions. Let $\#BIN$ the number of binaries after `tactic` and

`prime` simplification and reduction to REQ, that is the initial size of `allreq` in the algorithm. We know that the number of levels ($\#level$) is less than the minimum of $\#REQ$ and $\#BIN$. We observe that the real level is generally smaller than this limit but it seems difficult to predict it.

6.2 Simpler problems first

In fact, we observe that in our examples the problems are located near the top of the enumerating process. Of course, it is not difficult to find artificial examples where this is not the case and we can build examples containing only one problem situated at the last level.

Listing 1.6. Healthcare results

```
### level 1 problems ---------------
And(Doctor(T, X), Not(PrimaryDoctor(T, X)))
And(Patient(T, X), PrimaryDoctor(T, X))
And(Doctor(T, X), Receptionist(T, X))
And(Nurse(T, X), Doctor(T, X))
And(Manager(T, X), assign(T, X, Y))
And(Doctor(T, X), Not(assign(T, X, Y)), revoke(T, X, Y),
    ForAll(Q, And(Q < T, assign(Q, X, Y))))

### level 2 problems ---------------
And(Doctor(T, X), Not(PrimaryDoctor(T, X)))
And(Patient(T, X), PrimaryDoctor(T, X))
And(Doctor(T, X), Receptionist(T, X))
And(Nurse(T, X), Doctor(T, X))
And(Manager(T, X), assign(T, X, Y))
And(Doctor(T, X), Not(assign(T, X, Y)), revoke(T, X, Y),
    ForAll(Q, And(Q < T, assign(Q, X, Y))))
And(Doctor(T, X), Patient(T, X))

### level 3 problems ---------------
### <same problems as above>
```

Most of the problems are found after the initial steps or after the binary combinations. If there is only few problems no simplification is possible but if we have more, the simplified problems go up in the hierarchy of requests and are found first. Let K the maximal number of atoms in the problems, that is the number of bits different of -1. Initially, the reductions have a minimal number of atoms which is greater or equal to one. At each step the combining process combines distinct reductions and this increases the number of atoms at least of one. Since this computation is exhaustive we will reach the maximum K at most after $K-1$ steps. Listing 1.6 shows the three first levels for the Healthcare case study. We get 7 cases but the 6th is an unsatisfiable request and should be avoided, most of the problems are found by the initial step. We do not know how to exactly evaluate K, but one interesting heuristic is to observe the first level where no new problem is computed. The results are shown in Table 5, where $\#SL$ is the first level where no new problem appears.

It can be noticed that the time to process and the number of checks are radically smaller due to the reduced level of combinations. We should check with the solver the logical equivalence between the two sets of problems, but in fact they are syntactically the same.

Name	*time* (s)	*#problems*	*#checking*	*#REQ*	*#BIN*	*#SL*
RBSSIM	17	13	3688	32	31	3
Healthcare	45	6	2216	23	37	3
ContinueA	32	64	5968	30	42	3

Table 5. Results with the heuristic

6.3 Discussion

Due to lack of information our specifications are missing some relevant relations. For instance, in the Healthcare case we may assume that all resources are disjoint. This adds 78 explicit unsafe rules and as many problems. This multiplies the number of rules by two but this does not penalize the `combine` algorithm. Explicit unsafe with atoms in *REQ* are problems which are detected by the initial step. This increases the success rate for a combination to be already included in a problem thus less combinations are checked by the solver. With this new specification we have 139 rules and we get 84 problems in 34 seconds. We also note that the sufficient level is reached at level 2.

There are many different algorithms and optimizations, we defined and studied only some of them. For instance, we known that once a conjunction is defined all its contained conjunctions are also defined. Thus in the above algorithm we can bypass the check by the solver replacing it with a syntactic check that the new combination includes some already seen combinations. This is convenient if the syntactic check is faster than the solver execution. And in fact this is not obvious, except using a prime integer encoding which slightly improves the performances for the Healthcare and the ContinueA cases.

Another example is based on the fact that some enumerated combinations are not useful since they are included in already known problems. Though, adding such a check does not have a decisive impact on performances mainly because checking that the binary has been already seen is simpler and more general. With this optimization, ContinueA and Healthcare perform slightly better, RBSSIM slightly worst.

We do not study a smart tactic for the negation of the rule system but choosing a tactic which computes undefined requests with a maximum of atoms would increase the convergence of the above process.

It seems difficult to predict the value of the sufficient level to reach all the problems. One idea was to consider the set of problems and the set of undefined requests which are not problems. Another track is to explore the maximal satisfiable combinations of the existing combinations and to check if they are making explicit new problems or not. An alternative is to bound the K value from the rule system.

In our experiment we consider two assumptions about the choice of *REQ*. The first is to consider as atoms the expressions at the top and under a Boolean operator. Relaxing this assumption would multiply the number of atoms and have a bad impact on performances. The second choice was to consider that an

atom or its negation are allowed in the correct requests. It is simple to relax this assumption by considering the set of binary requests which are correct. This would have little impact on the performance of the last algorithm.

7 Conclusion

It is obvious that finding all the problems of a rule system is a computationally expensive task. There are three main reasons: the input request enumeration, the solver checking and the simplification process. We have discussed several possible processes and shown one complete algorithm but with still poor performances. This algorithm produces the simpler and abstract problems first, which can be considered as the most critical problems. Thus it provides a sensible practical tooling to analyze a rule system. However, one interesting heuristic is possible and allows us to compute most of the problems (if not all) in a reasonable amount of time.

As future work we plan to investigate ways of configuring the searching process and to exploit the simplified list of problems to further assist an iterative problem-fixing activity.

References

1. Aqib, M., Shaikh, R.A.: Analysis and comparison of access control policies validation mechanisms. I.J. Computer Network and Information Security **7**(1), 54–69 (2015)
2. Breaux, T.D., Hibshi, H., Rao, A.: Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements. Requir. Eng **19**(3), 281–307 (2014), `http://dx.doi.org/10.1007/s00766-013-0190-7`
3. Cau, A., Janicke, H., Moszkowski, B.C.: Verification and enforcement of access control policies. Formal Methods in System Design **43**(3), 450–492 (2013). https://doi.org/10.1007/s10703-013-0187-3
4. Cheng, Z., Royer, J.C., Tisi, M.: Efficiently characterizing the undefined requests of a rule-based system. In: Furia, C.A., Winter, K. (eds.) Integrated Formal Methods. LNCS, vol. 11023, pp. 69–88. Springer (2018)
5. Cheng, Z., Royer, J.C., Tisi, M.: Removing problems in rule-based policies. In: Dhillon, G., Karlsson, F., Hedström, K., Zúquete, A. (eds.) ICT Systems Security and Privacy Protection. pp. 120–133. Springer International Publishing (2019)
6. Coenen, F., Eaglestone, B., Ridley, M.J.: Verification, validation, and integrity issues in expert and database systems: Two perspectives. Int. J. Intell. Syst **16**(3), 425–447 (2001)
7. Coudert, O.: Two-level logic minimization: An overview. Integr. VLSI J. **17**(2), 97–140 (Oct 1994)
8. Craven, R., Lobo, J., Ma, J., Russo, A., Lupu, E.C., Bandara, A.K.: Expressive policy analysis with enhanced system dynamicity. In: Li, W., Susilo, W., Tupakula, U.K., Safavi-Naini, R., Varadharajan, V. (eds.) Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security. pp. 239–250. ACM (2009). https://doi.org/http://doi.acm.org/10.1145/1533057.1533091

9. Garcia-Alfaro, J., Cuppens, F., Cuppens-Boulahia, N., Martinez, S., Cabot, J.: Management of stateful firewall misconfiguration. Computers and Security **39**, 64–85 (2013)
10. Halpern, J.Y., Weissman, V.: Using first-order logic to reason about policies. ACM Transactions on Information and System Security **11**(4), 1–41 (Jul 2008). https://doi.org/http://doi.acm.org/10.1145/1380564.1380569
11. Han, W., Lei, C.: A survey on policy languages in network and security management. Computer Networks **56**(1), 477–489 (Jan 2012)
12. Hanamsagar, A., Jane, N., Borate, B., Wasvand, A., Darade, S.: Firewall anomaly management: A survey. International Journal of Computer Applications **105**(18), 1–5 (2014)
13. Hu, H., Ahn, G.J., Kulkarni, K.: Discovery and resolution of anomalies in web access control policies. IEEE Transactions on Dependable and Secure Computing **10**(6), 341–354 (2013)
14. Hwang, J., Xie, T., Hu, V.C.: Detection of multiple-duty-related security leakage in access control policies. In: Secure Software Integration and Reliability Improvement. pp. 65–74. IEEE Computer Society (2009), `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5325010;http://www.computer.org/csdl/proceedings/ssiri/2009/3758/00/index.html`
15. Liffiton, M.H., Malik, A.: Enumerating Infeasibility: Finding Multiple MUSes Quickly. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 160–175. Springer (2013)
16. Monperrus, M.: Automatic software repair: A bibliography. ACM Computing Surveys **51**(1), 17:1–17:24 (2018)
17. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis (TACAS). LNCS, vol. 4963, pp. 337–340. Springer-Verlag, Berlin (2008)
18. Nalepa, G.: A new approach to the rule-based systems design and implementation process. Computer Science **6**(5), 65 (2004). https://doi.org/10.7494/csci.2004.6.5.65, `https://journals.agh.edu.pl/csci/article/view/496`
19. Ni, Q., Bertino, E., Lobo, J., Brodie, C., Karat, C.M., Karat, J., Trombeta, A.: Privacy-aware role-based access control. ACM Trans. Inf. Syst. Secur. **13**(3), 24:1–24:31 (Jul 2010). https://doi.org/10.1145/1805974.1805980
20. Paschke, A.: Verification, validation and integrity of distributed and interchanged rule based policies and contracts in the semantic web. In: Semantic Web, 2nd International Semantic Web Policy Workshop (SWPW'06). CEUR-WS.org (2006)
21. Previti, A., Marques-Silva, J.: Partial MUS Enumeration. In: 27th AAAI Conference on Artificial Intelligence. pp. 818–825. AAAI Press, Bellevue, Washington (2013)
22. Royer, J.C.: The Conflict Notion and its Static Detection: a Formal Survey. Research report, IMT Atlantique (Jul 2019), `https://hal.archives-ouvertes.fr/hal-02169360`
23. Son, S., McKinley, K.S., Shmatikov, V.: Fix me up: Repairing access-control bugs in web applications. In: 20th Annual Network and Distributed System Security Symposium. Usenix, San Diego, California, USA (2013)
24. Wu, H.: Finding achievable features and constraint conflicts for inconsistent metamodels. In: 13th European Conference on Modelling Foundations and Applications. pp. 179–196. Springer, Marburg, Germany (2017)

25. Zacharias, V.: Development and verification of rule based systems - a survey of developers. In: Bassiliades, N., Governatori, G., Paschke, A. (eds.) RuleML 2008. LNCS, vol. 5321, pp. 6–16. Springer (2008)