Consistency Verification of Specification Rules

Thai Son Hoang, Shinji Itoh(✉), Kyohei Oyama, Kunihiko Miyazaki, Hironobu Kuruma, and Naoto Sato

Center for Technology Innovation, Hitachi Ltd.,
Yokohama, Kanagawa 244-0817, Japan
{hoang.thaison.ex,shinji.itoh.wn,kyohei.oyama.ec,kunihiko.miyazaki.zt, hironobu.kuruma.zg,naoto.sato.je}@hitachi.com

Abstract. This paper focuses on the consistency analysis of specification rules expressing relationships between input and expected output of systems. We identified the link between Minimal Inconsistent Sets (MISes) of rules and Minimal Unsatisfiable Subsets (MUSes) of constraints. For practical consistency verification of rules, we developed a novel algorithm using SMT solvers for fast enumeration of MUSes. We evaluated the algorithm using publicly available benchmarks. Finally, we used the approach to verify the consistency of specifications rules extracted from real-world case studies.

Keywords: Specification rules · Consistency verification · Minimal Inconsistent Sets (MISes) · Minimal Unsatisfiable Subsets (MUSes) · SMTs

1 Introduction

In financial and public sectors, regulations and policies are often specified in terms of rules *describing relationships between input and expected output*. As an example, consider a rewarding policy for a vehicle insurance company. Beside the normal contracts, the company offers two special rewards in the form of *discounts* (in percentages) for the insurance and shopping *coupons*. The availability of the rewards to customers depends on the *duration* (number of years) of the contracts, their online *account* status (whether or not they already have an online account), and their *VIP* membership status. The policies on how rewards are offered to a customer are as follows.

(R1) If the customer has an online account then either a discount of 3 % or a 100$ coupon is offered.
(R2) If the customer is a VIP then a discount of at least 5 % and a coupon valued between 50$ and 100$ are offered.
(R3) If the customer is not a VIP and the contract duration is less than 2 years then either a discount of less than 5 % or a coupon valued between 30$ and 50$ is offered.
(R4) If the customer is a VIP and the duration of the contract is at least 2 years then a discount of at least 7 % and a 50$ coupon are offered.

M. Butler et al. (Eds.): ICFEM 2015, LNCS 9407, pp. 50–66, 2015.
DOI: 10.1007/978-3-319-25423-4_4

Each rule comprises a constraint on the input and a constraint on the output, imitating logical implication. In particular, the rules are *non-deterministic*, *i.e.*, given a rule and some input, there could be more than one satisfying output. This type of *specification rules* is particularly useful in the early system design process, where requirements are obscure and the system details cannot be decided. Non-determinism is essentially what makes specification rules different from *production rules* used in *production rules used in Business Rule Management Systems* (BRMSes) [7]. Production rules are designed for execution, and hence they are necessarily deterministic. More discussion on the similarities and differences between specification rules and production rules can be seen in Sect. 6.

Given a set of rules, *i.e.*, a rule base, various properties can be statically analysed. One of the most important properties of a rule base is *consistency*: the rule base should be conflict-free, *i.e.*, there must be some possible output for any valid input. Otherwise, the regulations represented by the rule base are infeasible and cannot be implemented. In the example of the vehicle insurance company, the policy is inconsistent. More specifically, when a customer is a VIP with a 3-year contract and an online account, there are no possible values for the insurance discount and the shopping coupon satisfying the rewarding policy.

This paper focuses on the *consistency analysis* of a special type of specification rules, namely those where the output constraints do not refer to the input (*e.g.*, the vehicle insurance example). This type of specification rules is sufficient to stipulate many policies in financial and public sectors such as taxation regulations or insurance policies. Consistency analysis of this type of rules is a challenging problem. In the worst cases, there can be exponentially many set of inconsistent rules within a rule base. Moreover, even in the case where the rule base is consistent, one (potentially) has to consider every combination of the rules. Our motivation is to develop some program for *efficiently validating the consistency of specification rules*.

Within our knowledge, there are no existing technologies for formally verifying consistency of non-deterministic specification rules. However, given the fact that each rule is made up of an input constraint and an output constraint, the consistency of rules is related to the *satisfiability* of constraints. Recent advancement in the field of SMT solvers enables the possibility of checking satisfiability for a large and complex set of constraints of different types [4]. In particular, SMT solvers have been showed to be applicable to hardware designs, programs verification, etc. Various SMT-based problems have been investigated. Amongst them is "infeasibility analysis", the study about constraint sets for which no satisfying assignments exist. Given an unsatisfiable constraint set, useful information about this set includes to identify where the "problem" occurs. There exist efficient algorithms for extracting a *Minimal Unsatisfiable (sub-)Set* (MUS), i.e., the *unsat-core*, of an unsatisfiable constraint set [6,13,15]. Recently, algorithms for finding all MUSes have been proposed [3,10,11].

In order to validate the consistency of a rule base, we enumerate all *Minimal Inconsistent Sets* (MISes) of the rule base. A MIS is a set of inconsistent rules

that is minimal, with respect to the set-inclusion ordering. Similar to the MUSes of constraints, the MISes of rules identify where the problems occur within the rule base. By exploring the relationship between the MISes of the rule base, the MUSes of the output constraints, and satisfiability of individual input constraint, we reduce the problem of enumerating the MISes to that of the MUSes of the output constraints. We use SMT solvers as black-boxes for solving satisfiability problems. Furthermore, our approach is constraint-agnostic, *i.e.*, independent of the type of the input and output constraints.

We identify the relationship between the MISes of the rules, the MUSes of the output constraints, and the satisfiability of the input constraints. Our contribution is a novel algorithm for fast enumeration of MUSes. We compare our algorithm against the state-of-the-art program for MUSes enumeration from [10] using some publicly available benchmarks. The correctness of our approach is ensured by the formalisation of the algorithms using the Event-B modelling method [1] and the mechanical proofs using the supporting Rodin platform [2]. A more detailed version of this paper including the Event-B formalisation can be found elsewhere [9].

The rest of the paper is structured as follows. In Sect. 2, we present some background information including the problem of constraints satisfiability and rules consistency. In Sect. 3, we discuss the relationship between MISes and MUSes, showing that the problem of finding MISes can be reduced to enumerating MUSes. In Sect. 4, we present a novel and efficient algorithm for enumerating MUSes. In Sect. 5, we give our empirical analysis of the new algorithm and its application in finding MISes. Finally, we draw some conclusions in Sect. 6.

2 Background

2.1 Constraints Satisfiability

In this paper, we often discuss satisfiability problems related to different generic sets of constraints. For each set of constraints, the constraint type and variables domain are omitted. In general, we will consider some indexed set of constraints $\mathbb{C} = \{C_1, C_2, \ldots, C_n\}$. Each constraint C_i specifies some restrictions on the problem's variables. Constraint C_i is satisfied by any assignment $\mathbf{A}$ of the variables that meets C_i's restriction. We use the notation $\mathsf{sat}(\mathbf{A}, C)$ to denote the fact that $\mathbf{A}$ satisfies C, and $\mathsf{unsat}(\mathbf{A}, C)$ otherwise.

Given a set of constraints $Cs \subseteq \mathbb{C}$, if there exists some assignment satisfying every constraint in Cs then Cs is said to be satisfiable (SAT). Otherwise, Cs is unsatisfiable (UNSAT). More formally, given a set of constraints Cs, we have

$$\mathsf{SAT}(Cs) \;\widehat{=}\; \exists \mathbf{A} \cdot \forall C \in Cs \cdot \mathsf{sat}(\mathbf{A}, C), \text{ and} \tag{1}$$

$$\mathsf{UNSAT}(Cs) \;\widehat{=}\; \forall \mathbf{A} \cdot \exists C \in Cs \cdot \mathsf{unsat}(\mathbf{A}, C). \tag{2}$$

In this paper, we will be interested in two special types of sets of constraints, namely: *Maximal Satisfiable (sub-)Set* (MSS) and *Minimal Unsatisfiable*

(sub-)Set (MUS). A set of constraints Cs is an MSS if it is a satisfiable subset of $\mathbb{C}$ and cannot be expanded without compromising satisfiability, *i.e.*,[1]

$$\mathsf{MSS}(Cs) \mathrel{\widehat{=}} \mathsf{SAT}(Cs) \wedge (\forall S \cdot S \subseteq \mathbb{C} \wedge Cs \subset S \Rightarrow \mathsf{UNSAT}(S)). \tag{3}$$

Conversely, a set of constraints $Cs \subseteq \mathbb{C}$ is a MUS if it is an unsatisfiable subset of $\mathbb{C}$ and is minimal with respect to the set-inclusion ordering, *i.e.*,

$$\mathsf{MUS}(Cs) \mathrel{\widehat{=}} \mathsf{UNSAT}(Cs) \ \wedge \ (\forall S \cdot S \subseteq \mathbb{C} \wedge S \subset Cs \Rightarrow \mathsf{SAT}(S)). \tag{4}$$

MUSes are valuable since they indicate the core reason for unsatisfiability of a constraint set. In particular, as showed in Sect. 3, MUSes play an important role in verifying rules consistency.

2.2 Rules Consistency

Consider a generic set of rules $\mathbb{R} = \{R_1, R_2, \ldots, R_n\}$, where n is a positive number. Each rule R_i consists of a constraint I_i over the input variables and a constraint O_i over the output variables. The set of input and output variables are disjoint. The types of constraints are not specified.

Definition 1 (Rule Satisfiability). *A rule $R = (I, O)$ is satisfied by an assignment $\mathbf{A}_x$ of the input variables and an assignment $\mathbf{A}_y$ of the output variables —denoted as $\mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R)$— if either $\mathbf{A}_x$ does not satisfy I or $\mathbf{A}_y$ satisfies O, i.e., $\mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R) \mathrel{\widehat{=}} \mathsf{unsat}(\mathbf{A}_x, I) \vee \mathsf{sat}(\mathbf{A}_y, O)$.*

A subset of rules $Rs \subseteq \mathbb{R}$ is "consistent" (Consistent) if for every input assignment, there exists some output assignment such that the assignments satisfy all rules in Rs. Otherwise, it is inconsistent (Inconsistent). For convenience, when the generic set of rules $\mathbb{R}$ is known, we identify its subsets by sets of indices, *i.e.*, subsets of the range $1..n$. The consistency definition is lifted accordingly to sets of indices.

Definition 2 (Rule Consistency). *Given a set of indices $S \subseteq 1..n$,*

$$\mathsf{Consistent}(S) \ \widehat{=} \ \forall \mathbf{A}_x \cdot \exists \mathbf{A}_y \cdot \forall i \in S \cdot \mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R_i), \ \textit{and} \tag{5}$$

$$\mathsf{Inconsistent}(S) \ \widehat{=} \ \exists \mathbf{A}_x \cdot \forall \mathbf{A}_y \cdot \exists i \in S \cdot \neg \mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R_i). \tag{6}$$

From now on, we will use *set of rules* and *set of rule indices* interchangeably.

Given an inconsistent rule base $\mathbb{R}$, some indicating facts about $\mathbb{R}$ should be given to "explain" $\mathbb{R}$'s inconsistency. We define the following notion of *Minimal Inconsistent Set* (MIS) of a set of rules, the *inconsistent core* of $\mathbb{R}$.

Definition 3 (MIS). *Given a set of rules $S \subseteq 1..n$, S is a MIS if and only if S is inconsistent and is minimal with respect to the set-inclusion ordering, i.e., $\mathsf{MIS}(S) \mathrel{\widehat{=}} \mathsf{Inconsistent}(S) \wedge (\forall T \cdot T \subset S \Rightarrow \mathsf{Consistent}(T))$.*

[1] $S \subset T$ means that S is a proper-subset of T.

Clearly, a rule base $\mathbb{R}$ without any MIS is consistent. In the case where $\mathbb{R}$ is inconsistent, ideally, all MISes of $\mathbb{R}$ should be found. In general, the problem of finding all MISes is intractable: the number of MISes may be exponential in the size of the rule base. Our main objective is to *quickly enumerate MISes*.

Example 1. Consider the rewarding policy mentioned in Sect. 1. The input and output constraints of the rules can be formalised as follows.

Rule	Input constraint	Output constraint
R_1	$account$	$discount = 3 \vee coupon = 100$
R_2	VIP	$discount \geq 5 \wedge coupon \in 50 \,..\, 100$
R_3	$\neg VIP \wedge duration < 2$	$discount < 5 \vee coupon \in 30 \,..\, 50$
R_4	$VIP \wedge duration \geq 2$	$discount \geq 7 \wedge coupon = 50$

In the above example, input assignment "$account, VIP, duration := \mathsf{F}, \mathsf{T}, 1$" and output assignment "$discount, coupon := 5, 50$" satisfy all rules. The set of rules is inconsistent (as mentioned before) and has one MIS, *i.e.*, $\{R_1, R_4\}$.

3 Relationship Between MISes and MUSes

In this section, we investigate the relationship between MISes and satisfiability problems on the input and output constraints. Since the input and output variables are disjoint, satisfiability problems on input constraints and output constraints are independent. Given a set of rules S, the following lemmas express some relationships between the consistency of S and the satisfiability of S's input and output constraints. The lemmas can be proved directly from the corresponding definitions. Below, we use the notation $C[S]$ for $\{C_i \mid i \in S\}$ (similarly for $I[S]$ and $O[S]$).

Lemma 1. $\mathsf{SAT}(I[S]) \wedge \mathsf{UNSAT}(O[S]) \;\Rightarrow\; \mathsf{Inconsistent}(S)$

Lemma 2. $\mathsf{SAT}(O[S]) \;\Rightarrow\; \mathsf{Consistent}(S)$

In general, S's consistency cannot be directly determined by the satisfiability/unsatisfiability of its input and output constraints. In particular, when $\mathsf{UNSAT}(I[S])$ and $\mathsf{UNSAT}(O[S])$, S's consistency is determined by the consistency of S's proper-subsets. Consider Example 1, $\{R_1, R_2, R_3\}$ and $\{R_1, R_3, R_4\}$ have unsatisfiable input and output constraints, but only the former one is consistent. To avoid iterating the subsets of rules, we prove the following theorem.

Theorem 1 (MISes and MUSes). $\mathsf{MIS}(S) \;\Leftrightarrow\; \mathsf{MUS}(O[S]) \wedge \mathsf{SAT}(I[S])$.

Proof (Sketch).

1. *From left to right*: Assume $\mathsf{MIS}(S)$, we have $\mathsf{Inconsistent}(S)$. We infer that $\mathsf{SAT}(I[S])$ since if $\mathsf{UNSAT}(I[S])$, one of S's proper-subset must be inconsistent, hence S cannot be minimal. We have $\mathsf{UNSAT}(O[S])$ since if $\mathsf{SAT}(O[S])$ then $\mathsf{Consistent}(S)$ (Lemma 2). Subsequently, $\mathsf{MUS}(O[S])$, since otherwise there exists a set $T \subset S$ such that $\mathsf{UNSAT}(O[T])$. From Lemma 1, we have $\mathsf{Inconsistent}(T)$, and hence S is not minimal inconsistent.

2. *From right to left*: Assume MUS($O[S]$) and SAT($I[S]$). We deduce that Inconsistent(S) from Lemma 1. For every $T \subset S$, we have SAT($O[T]$), hence Consistent(T) (Lemma 2). As a result, S is minimal inconsistent. □

Theorem 1 reduces the problem of enumerating MISes to finding the MUSes of the output constraints, and then checking the satisfiability of the input constraints corresponding to each MUS found. As a result, the quicker output constraints' MUSes are discovered, the faster we can enumerate MISes. In the next section, we present a novel and efficient algorithm for enumerating MUSes.

4 An Efficient Algorithm for Enumerating MUSes

In general, enumerating MUSes is a well-known problem and potentially intractable (since the number of MUSes may be exponential in the number of constraints). A detailed discussion on existing approaches for enumerating MUSes can be found in [10], including both constraint-specific and constraint-agnostic algorithms. In this paper, we present our algorithm for fast enumeration of MUSes, inspired by the state-of-the-art algorithm MARCO [10].

The main feature of MARCO is the use of a powerset manager maintaining a powerset map for selecting subsets to be explored. Given, a constraint set $\mathbb{C}$, the powerset map is a set of propositions Ps over a collection of indexed variables x_i, with $i \in 1..n$ where n is the number of constraints in $\mathbb{C}$. There are three basic operations for the powerset manager (Fig. 1). In getSet, the powerset manager utilises the capability of the constraint solver to return a model for a satisfiable set of constraints, which corresponds to an unexplored subset (a subset required to be validated). Operations addLowerBound and addUpperBound are for pruning the unexplored subsets. Operation addLowerBound (resp. addUpperBound) marks all subsets (resp. supersets) of the input set S as explored (no longer need to be validated).

getSet $\widehat{=}$
output: a new unexplored subset or *null* if all subsets have been explored.

```
1. if SAT(Ps)                          // If there are some unexplored subset, then
2.    m ← getModel(Ps);                //    get a model
3.    return {i | m(x_i) = True};      //    return the unexplored subset from the model
4. else                                // If all subsets have been explored, then
5.    return null;                     //    return null
```

addLowerBound(S) $\widehat{=}$
precondition: $S \subseteq 1\,..\,n$
effect: Mark subsets of S explored

1. $Ps := Ps \cup \{(\bigvee_{i \notin S} x_i)\}$;

addUpperBound(S) $\widehat{=}$
precondition: $S \subseteq 1\,..\,n$
effect: Mark supersets of S explored

1. $Ps := Ps \cup \{(\neg \bigwedge_{i \in S} x_i)\}$;

Fig. 1. Operations of the powerset manager

4.1 The MARCO algorithm

Intuitively, for each iteration, MARCO (Fig. 2) gets a new unexplored subset S from the powerset manager. If $\mathbb{C}[S]$ is satisfiable, MARCO uses a grow subroutine to obtain an MSS, and adds this MSS as a lower bound to restrict future iterations. Otherwise, *i.e.*, if $\mathbb{C}[S]$ is unsatisfiable, MARCO uses a shrink subroutine to obtain a MUS, yields this MUS, and adds the found MUS as an upper bound. The correctness of the MARCO algorithm relies on the fact that MUS cannot be a subset of an MSS or a (strict-)superset of another MUS. At each iteration, the powerset manager is restricted hence the algorithm terminates.

```
MARCO ≙
effect: Output MUSes of ℂ as they are found
1.  S ← getSet();                  // S is an unexplored subset
2.  while S ≠ null                 // While there is some unexplored subset S,
3.    if SAT(ℂ[S])                 //   if S is satisfiable,
4.      mss ← grow(S);             //     grow S to obtain an MSS
5.      addLowerBound(mss);        //     add the found mss as a lower bound
6.    else                         //   if S is unsatisfiable,
7.      mus ← shrink(S);           //     shrink S to obtain a MUS
8.      yields mus;                //     yields the found MUS
9.      addUpperBound(mus);        //     add the found mus as an upper bound
10.   S ← getSet();                // Get a new unexplored subset S
```

Fig. 2. The MARCO algorithm

The sub-routines grow and shrink can be any off-the-shelf methods for finding an MSS (from a satisfiable seed) and a MUS (from an unsatisfiable seed). For example, the operation growLin in Fig. 3 gradually adds new elements to a satisfiable subset S if this preserves satisfiability. Conversely, shrinkLin removes elements step-by-step from an unsatisfiable subset S if it preserves unsatisfiability. Both growLin and shrinkLin are not the most efficient implementation for grow and shrink sub-routine. For instance, the shrinkBin operation in Fig. 3 and its sub-routine reduce perform binary search and potentially return a MUS faster than shrinkLin. A similar binary search algorithm exists for the grow routine. The MARCO algorithm and various grow and shrink routines are not novel.

Example 2. A possible execution trace for MARCO (using growLin and shrinkBin) applied to the *output constraints* of the rules R_1–R_4 in Example 1 is below. At each step, we show the seed obtained from the powerset manager and its satisfiability status. Depending on the satisfiability status of the seed, a growing or a shrinking sub-routine is called to obtain an MSS or a MUS. We also report the number of SMT calls (the number of times that the SMT solver is called to check the problem constraints) and the number of SAT calls (querying the powerset manager). For example, in Step 2, the powerset manager returns $\{R_3, R_4\}$ (at a cost of 1 SAT call) and checking satisfiability of this seed costs 1 SMT call. Afterwards, it takes 2 SMT calls to grow the seed to obtain the MSS $\{R_2, R_3, R_4\}$.

```
growLin(S) ≙
precondition: SAT(ℂ[S])
return: an MSS of ℂ
1. foreach i ∉ S
2.    if SAT(ℂ[S ∪ {i}])
3.       S := S ∪ {i};
4. return S;
```

```
shrinkLin(S) ≙
precondition: UNSAT(ℂ[S])
return: a MUS of ℂ
1. foreach i ∈ S
2.    if UNSAT(ℂ[S \ {i}])
3.       S := S \ {i};
4. return S;
```

```
shrinkBin(S) ≙
precondition: UNSAT(ℂ[S])
return: a MUS of ℂ
1. return reduce(S, ∅);
```

```
reduce(A, B) ≙
precondition: UNSAT(ℂ[A ∪ B])
return: a minimal a such that
        a ⊆ A ∧ UNSAT(ℂ[a ∪ B])
1. C := A/2;
2. if UNSAT(ℂ[C ∪ B])
3.    return reduce(C, B);
4. D := A \ C;
5. if UNSAT(ℂ[D ∪ B])
6.    return reduce(D, B);
7. C1 ← reduce(C, D ∪ B);
8. D1 ← reduce(D, C1 ∪ B);
9. return C1 ∪ D1;
```

Fig. 3. Various grow and shrink routines

Step		Seed (Satisfiability status)	MSS	MUS	SMTs	SATs
1	get seed	$\varnothing$ (SAT)			1	1
	growing		$\{R_1, R_2\}$		4	
2	get seed	$\{R_3, R_4\}$ (SAT)			1	1
	growing		$\{R_2, R_3, R_4\}$		2	
3	get seed	$\{R_1, R_3, R_4\}$ (UNSAT)			1	1
	shrinking			$\{R_1, R_4\}$	4	
4	get seed	$\{R_1, R_2, R_3\}$ (UNSAT)			1	1
	shrinking			$\{R_1, R_2, R_3\}$	4	
5	get seed	$\{R_1, R_3\}$ (SAT)			1	1
	growing		$\{R_1, R_3\}$		2	
6	get seed	*null*				1
	Total		3 MSSes	2 MUSes	21	6

4.2 The MUSesHunter Algorithm

Notice that, in the MARCO algorithm, the powerset manager is only used for retrieving unexplored subsets. In particular, during the process of growing and shrinking seeds, many satisfiability checks are spurious: the sets are either supersets of some found MUS or subsets of some found MSS. Satisfiability checking of the problem constraints (possibly involving theories) is more expensive than querying the powerset manager (concerning only Boolean constraints). Furthermore, during the process of growing, often unsatisfiable subsets are found. By calling shrink sub-routine on these unsatisfiable subsets, we can get MUSes faster. The challenge is to ensure that by shrinking immediately, we obtain a *new* MUS.

The MUSesHunter algorithm can be seen in Fig. 4. Compared to MARCO, the main difference is the use of the powerset manager within the growHyb

and shrinkBinPS sub-routines. In particular, growHyb returns either a MUS or an MSS. In the subsequent, we outline the implementation of shrinkBinPS and growHyb. First, we extend the powerset manager with satisfiability checking.

```
MUSesHunter ≙
effect: Output MUSes of ℂ as they are found
1.  S ← getSet();                    // S is an unexplored subset
2.  while S ≠ null                   // While there is some unexplored subset S
3.    if SAT(ℂ[S])                   //   if S is satisfiable
4.      set ← growHyb(S);            //     grow/shrink S to have an MSS or a MUS
5.      if set is an MSS             //     if set is an MSS
6.        addLowerBound(set)         //       Add set as a lower bound
7.      else                         //     if set is a MUS
8.        yields set;                //       yields set as a new MUS
9.        addLowerBound(set);        //       add set as a lower bound
10.       addUpperBound(set);        //       add set as an upper bound
11.   else                           //   if S is unsatisfiable
12.     mus ← shrinkBinPS(S);        //     shrink S to obtain a MUS
13.     yields mus;                  //     yields mus as a new MUS
14.     addUpperBound(mus);          //     Add the found mus as an upper bound.
15.     addLowerBound(mus);          //     Add the found mus as a lower bound.
16.   S ← getSet();                  //   Get a new unexplored subset S
```

Fig. 4. The MUSesHunter algorithm

Satisfiability checking with powerset manager. The following operation is added to the powerset manager in order to check if a set of constraint S need to be explored. This is done by checking the satisfiability of the set of propositions Ps together with the constraint representing S.

```
isUnexplored(S)  ≙
precondition: S ⊆ 1 .. n
output: ⊤ if S is an unexplored subset
1. return SAT(Ps ∪ {(⋀_{i∈S} x_i) ∧ (⋀_{i∉S} ¬x_i)});
```

Theorem 2 states an important property of MUSesHunter, in particular, for the explored subsets filtered out by the powerset manager.

Theorem 2 (Explored subsets of constraints). *For the* MUSesHunter *algorithm, given a subset of constraints S, we have*

$$\neg\mathsf{isUnexplored}(S) \Leftrightarrow \begin{array}{l} (\exists L \cdot \mathsf{SAT}(\mathbb{C}[L]) \wedge S \subseteq L) \\ \vee\ (\exists M \cdot \mathsf{MUS}(\mathbb{C}[M]) \wedge S \subset M) \\ \vee\ (\exists M \cdot \mathsf{MUS}(\mathbb{C}[M]) \wedge M \subseteq S). \end{array} \tag{7}$$

Proof (Sketch). This fact is trivial invariant of the MUSesHunter algorithm since the set of unexplored subsets can only be pruned in the following two cases:

1. An MSS is added as a lower bound.
2. An MUS is added as a lower bound and an upper bound.

As a result, a set S is explored if and only if either (a) S is a subset of an MSS (some satisfying set L), or (b) S is a subset of some MUS M, or (c) S is a superset of some MUS M. □

The following Lemmas are consequences of Theorem 2.

Lemma 3 (Satisfiability during shrink). *Given sets of constraints S and T, if* isUnexplored(S), $T \subseteq S$, *and* $\neg$isUnexplored(T), *then* SAT$(\mathbb{C}[T])$.

Proof. From $\neg$isUnexplored(T), apply Theorem 2, we have three cases as follows.

1. There exists L where SAT$(\mathbb{C}[L]) \wedge T \subseteq L$, we have SAT$(\mathbb{C}[T])$ trivially by anti-monotonicity of SAT.
2. There exists M where MUS$(\mathbb{C}[M]) \wedge T \subset M$, we have SAT$(\mathbb{C}[T])$ trivially by definition of MUS (4).
3. There exists M where MUS$(\mathbb{C}[M]) \wedge M \subseteq T$. From $T \subseteq S$, we obtain $M \subseteq S$. Apply Theorem 2, we have $\neg$isUnexplored(S) which is a contradiction.

Lemma 4 (Unsatisfiability during grow). *Given sets of constraints S and T, if* isUnexplored(S), $S \subseteq T$, *and* $\neg$isUnexplored(T), *then* UNSAT$(\mathbb{C}[T])$.

The proof of Lemma 4 is similar to that of Lemma 3 and is omitted.

Lemmas 3 and 4 allow us to use the powerset manager to replace some of the satisfiability checks during shrinking and growing sub-routines.

The shrinkBinPS *sub-routine.* Comparing the subsequent reducePS with the reduce sub-routine, before checking satisfiability of $C \cup B$ (Line 3) and $D \cup B$ (Line 6), we first check if these subsets are unexplored. Lemma 3 ensures that if they are already explored, they are satisfiable.

```
shrinkBinPS(S)  ≙
precondition: UNSAT(ℂ[S]) ∧ isUnexplored(S)
output: a MUS of Cs
1. return reducePS(S, ∅);
```

```
reducePS(A, B)  ≙
precondition: UNSAT(ℂ[A ∪ B]) ∧ isUnexplored(A ∪ B)
output: a minimal a ⊆ A ∧ UNSAT(ℂ[a ∪ B])
1. C := A/2;                          // C is a half of A
2. if isUnexplored(C ∪ B)             // If C ∪ B is unexplored,
3.   if UNSAT(ℂ[C ∪ B])               //   if C ∪ B is unsatisfiable,
4.     return reducePS(C, B);         //     recursively reduce C with B
5. D := A \ C;                        // D is the difference between A and C
6. if isUnexplored(D ∪ B)             // If D ∪ B is unexplored,
7.   if UNSAT(ℂ[D ∪ B])               //   if D ∪ B is unsatisfiable,
8.     return reducePS(D, B);         //     recursively reduce D with B
9. C1 ← reducePS(C, D ∪ B);           // C1 is the result of reducing C with D ∪ B
10. D1 ← reducePS(D, C1 ∪ B);         // D1 is the result of reducing D with C1 ∪ B
11. return C1 ∪ D1;                   // return the union of C1 and D1
```

The growHyb *sub-routine.* The following growHyb returns either a new MSS or a new MUS. It is based on the growLin routine showed earlier.

```
   growHyb(S)  ≙
   precondition: SAT(ℂ[S]) ∧ isUnexplored(S)
   return: an MSS or a MUS of ℂ
1. foreach c ∉ S                          // For each c not in S,
2.   if isUnexplored(S ∪ {c})            //   if S ∪ {c} is unexplored
3.     if SAT(ℂ[S ∪ {c}])                //     if S ∪ {c} is satisfiable,
4.       S := S ∪ {c};                   //       add c to S
5.     else                              //     if S ∪ {c} is unsatisfiable
6.       return shrinkBinPS(S ∪ {c});    //       shrink to find a MUS
7. return S;                              // return S which is an MSS
```

Similar to the reducePS sub-routine, before checking satisfiability for $S \cup \{c\}$ (Line 3), the growHyb sub-routine checks if it is unexplored. Lemma 4 ensures that if $S \cup \{c\}$ is already explored, it is unsatisfiable. Moreover, the fact that $S \cup \{c\}$ is unexplored guarantees that shrinkBinPS (Line 6) returns a *new MUS*.

Example 3. An example execution trace for the MUSesHunter algorithm (using growHyb and shrinkBinPS) applying to the set of output constraints for the rules R_1–R_4 in Example 1 is below.

Step		Seed (Status)	MSS	MUS	SMTs	SATs
1	get seed	$\varnothing$ (SAT)			1	1
	growing	$\{1,2,3\}$ (UNSAT)			3	3
	shrinking			$\{1,2,3\}$	2	4
2	get seed	$\{4\}$ (SAT)			1	1
	growing	$\{1,4\}$ (UNSAT)			1	1
	shrinking			$\{1,4\}$		2
3	get seed	$\{2,3,4\}$ (SAT)			1	1
	growing		$\{2,3,4\}$			1
4	get seed	*null*				1
	Total		1 MSSes	2 MUSes	9	15

4.3 Comparing MARCO and MUSesHunter

The main novelty of MUSesHunter compared to MARCO is the use of the powerset manager for checking satisfiability of the problem constraints. In particular, the powerset manager allows MUSesHunter to produce MUSes even in the case where the original seed is satisfiable, where MARCO must always produce MSSes. For our purpose of enumerating MUSes, finding a MUS is more valuable than MSS. When a MUS is found, MUSesHunter blocks both its super-sets as well as subsets, where MARCO only blocks the super-sets of MUS. As a result, MUSesHunter prunes the search space much faster than MARCO. Comparing the traces for MARCO and MUSesHunter in Examples 2 and 3, MUSesHunter does not need to find all MSSes before termination. In fact MSSes found by MARCO such as $\{R_1, R_2\}$ and $\{R_1, R_3\}$ are spurious, *i.e.*, they are subset of the MUS $\{R_1, R_2, R_3\}$, and does not require to be considered in searching for MUSes. For MARCO, MUS can be also added as a lower bound. Focusing on enumerating MUSes without finding all MSSes was mentioned as future work in [10].

5 Empirical Analysis

We implement our algorithms for finding MUSes and MISes using Java. In particular, for constraint solving (*i.e.*, for the powerset manager and for satisfiability checks of the problem constraints), we use SMTInterpol [8]. We first compare the performance of the MUSesHunter and MARCO algorithms (Sect. 5.1). Afterwards, we evaluate the performance of developed MISes finder program with MUSesHunter using examples extracted from real-world policies (Sect. 5.2).

5.1 MUSesHunter vs. MARCO

Both algorithms were implemented using the same underlying infrastructure, sharing as much code as possible. For growing and shrinking, MARCO uses growLin and shrinkBin sub-routines, whereas MUSesHunter uses growHyb and shrinkBinPS sub-routines. Both algorithms use a powerset manager built on top of SMTInterpol without any modification, *e.g.*, it is not biased towards producing large unexplored sets (which will be beneficial for MARCO). The experiments were performed on a VMWare Virtual Machine with 4×2.7 GHz CPUs running Linux. Each program was executed with 3 GB heap memory limit and an 1800-second timeout. There is no timeout for individual constraints satisfiability check. We selected 473 samples (from 4 to 881 constraints) selected from SMT-LIB for quantifier-free linear integer arithmetic (QF_LIA).[2] Even though our algorithm is constraint-agnostic, we have chosen the QF_LIA fragment of the SMT-LIB benchmarks since the input and output constraints of our rule verification case studies are within this sub-logic. We also restrict our evaluation to the sets of benchmarks where the SMT solver (SMTInterpol in our implementation) can verify their satisfiability in a reasonable time. We plan to evaluate our algorithm against other benchmarks in the future.

Overall. The summary of the results for running the two algorithms is in Table 1. While the numbers of cases where the algorithms terminate and find all MUSes are comparable, MUSesHunter tends to run out of memory (hitting bad seeds) whereas MARCO tends to run out of time (making too many expensive SMT calls) more often. However, in most cases, MUSesHunter usually finds more MUSes than MARCO. In particular, MARCO does not find any MUSes in over 20 % of the samples (105 cases), whereas that percentage for MUSesHunter is 6 % (21 cases). This is the direct effect of growHyb: it can produce MUSes even in the case where the original seed is satisfiable. On average, MUSesHunter found almost twice as many MUSes as MARCO.

Comparing the number of SMT calls for checking satisfiability of the problem constraints and SAT calls for the powerset manager, there is a clear difference between the two programs. MUSesHunter makes heavy use of the powerset manager as a substitute for checking satisfiability of the problem constraints. Even though MUSesHunter's total number of satisfiability calls is twice as many as that

[2] Available from http://smtlib.cs.uiowa.edu/benchmarks.shtml.

Table 1. Empirical analysis summary

	MUSesHunter		MARCO	
Find all (no. of samples)	160 (34%)		139 (29%)	
Timeout (no. of samples)	250 (53%)		308 (65%)	
Find none		21 (8%)		104 (34%)
Find 1		33 (13%)		76 (25%)
Find > 1		196 (78%)		128 (42%)
Out-of-memory (no. of samples)	63 (13%)		26 (5%)	
Find none		0 (0%)		1 (4%)
Find 1		9 (14%)		0 (0%)
Find > 1		54 (86%)		25 (96%)
Total (no. of samples)	473		473	
Max MUSes found per sample	29740		18646	
Average MUSes found per sample	1050		533	
Total satisfiability calls	7749472		3127773	
SATs (powerset manager)		6472339 (84%)		82628(3%)
SMTs (constraints satisfiability)		1277133 (16%)		3045145(97%)

of MARCO, solving satisfiability problems in the powerset manager (related to Boolean constraints) are much faster than checking satisfiability of the problem constraints, which are (in our experiments) QF_LIA constraints.

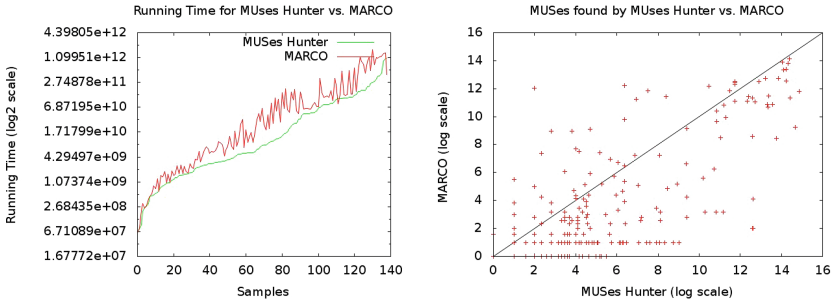
Performance comparison between MUSesHunter and MARCO is as follows.

	MUSesHunter better (%)	MARCO better (%)	Draw (%)
Both terminate	135 (97 %)	4 (3 %)	
Not both terminate	254 (76 %)	52 (16 %)	28 (8 %)
Total	389 (82 %)	56 (12 %)	28 (6 %)

We separate the benchmarks into two categories according to whether or not both algorithms terminate and find all MUSes. In the first case, an algorithm is better if it terminates faster. In the second case, we compare the number of MUSes that the algorithms found. Overall, MUSesHunter outperforms MARCO by terminating faster or finding more MUSes (82 %).

Both programs terminate. The comparison between the (log-scale) speed of MUSesHunter and MARCO in the case where they both terminate can be seen in Fig. 5a. In most cases (97 %), MUSesHunter terminates faster than MARCO. Figure 6 compares the percentage of MUSes found against the time, both scaled to the range 0..1 for samples that MUSesHunter and MARCO terminate. For MUSesHunter, it is typical that the MUSes are found early then subsequently, only MSSes are found. For MARCO, in most cases, MUSes are found gradually.

One of the programs does not terminate. We focus on the number of MUSes found by each algorithm. In 76 % of cases, MUSesHunter found more MUSes than MARCO. Figure 5b shows the comparison between the numbers of MUSes found by MUSesHunter and MARCO for individual sample.



(a) Running time comparison (b) MUSes found comparison

Fig. 5. Running time and MUSes found comparison

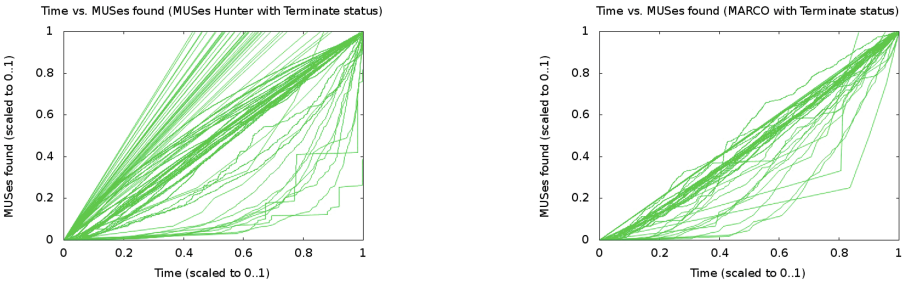


Fig. 6. MUSes found vs. time

5.2 MISes Finder

We evaluate the MISes finder program on the following examples. With the exception of the first one, all of them are extracted from industrial case studies in financial and public sectors. The statistics can be seen as follows.

Example	Size	Time	MUSes	MSSes	MISes	SATs	SMTs
Test sample	8	195 ms	5	12	3	78	57
Vehicle insurance	4	69 ms	2	1	1	9	15
Care insurance	15	403 ms	62	0	10	106	335
Vehicle tax	108	13.4 s	2590	2	0	7223	19345
Registration	725	1800 s	436	1093	0	820455	373767

The performance of the MISes finder program largely depend on the underlying MUSesHunter algorithm on finding MUSes of the output constraints. The last two examples (namely *Vehicle tax* and *Registration*) are consistent. However, MISes finder program does not terminate for the *Registration* example, fails to verify the rule base. In this case, all 436 MUSes (none of them are MISes)

are found within 60 s. Afterwards, the program only found MSSes. Given the size of the rule base, we do not expect the program found all MSSes within a reasonable time. To validate this set of rules, we need to adopt some additional techniques to reduce the complexity of the problem.

6 Conclusion

In this paper, we present our approach for validating the consistency of specification rules describing the relationship between the system input and expected output. Our method explores the relationship between MISes of rules and MUSes of constraints. We developed a novel algorithm for fast enumeration of MUSes during the validation process and evaluated it against MARCO [10], a state-of-the-art algorithm for enumerating MUSes. Our approach is constraint-agnostic and makes use of constraints solvers as black-boxes. Furthermore, we make use of the well-known routines such as shrink and grow sub-routines to find MUSes and MSSes. Any state-of-the-art implementation for these sub-routines can be used within our algorithm. Since our algorithm relying on SMT solvers for satisfiability checking, consistency verification of specifications rules will be limited by the capability of the underlying SMT solvers.

Related work. The similarity between specification rules and production rules [7] is in their composition: Each rule is composed of a guard (input constraints) and an action (output constraints). The main difference between them is the fact that specification rules can be non-deterministic while production rules are deterministic. Furthermore, the purpose of specification rules is to stipulate the relationships between input and output of the system, hiding the system state. The production rules often involve systems with explicit state (the working memory). Production rules are also written with some implicit rule execution engine in mind, *e.g.*, firing enabled rules repeatedly. In term of validation, different properties have been considered for production rules [14]. Due to their deterministic nature, minimal inconsistency for production rules can only be pairwise.

In a more general context, *Rule-Based Systems* (RBSes) have been used in the field of Artificial Intelligence and Knowledge Engineering [12] for knowledge representation. Typically, an RBS contains a rule set and an inference control mechanism including some conflict resolution strategy. Each rule imitates logical implication and is used for backward or forward reasoning. Furthermore the conflict resolution strategies ensure that in the case where two or more rules can be activated at a time, only one is selected according to some pre-defined criteria. This is also the main difference between RBSes and the specification rules under our consideration. Verification of RBSes is also extensively discussed in [12]. A taxonomy of verifiable characteristics for RBSes is proposed concerning various anomalies such as consistency, completeness, and (non-)redundancy. In particular, the problem of conflicting and inconsistent rules are considered to be special cases of nondeterminism and is deferred to the conflict resolution mechanism. In fact, in some special representation of RBSes such as tabular systems with no explicit negation, purely logical inconsistency never appears [12].

In order to find all MISes of a set of rules, we need to find all MUSes of its output constraints. Comparison in [10] suggests the CAMUS algorithm [11] can out-perform the MARCO algorithm in finding all MUSes. The disadvantage of CAMUS is its inability to "enumerate" the MUSes, *i.e.*, it can take a long time before outputting any MUS. Hence CAMUS is unsuitable for any application where incremental responses are required. However, its ability of finding all MUSes quickly can be useful to validate a consistent set of rules. We plan to investigate and evaluate CAMUS further.

Future work. Other properties for specification rules include *redundancy* and *completeness*. Similar to consistency, the problem of verifying redundancy and completeness can also be reduced to enumerating MUSes and this is the next logical step of our research. As mentioned before in Sect. 5.2, for the *Registration* example, our MISes finder program does not terminate. The main challenge is in the size of the example (725 rules). A possible solution for validating this set of rule is to syntactically decompose this set of rules into smaller sets. Rule separation will drastically reduce the complexity of the MISes finding problem, hence could be used as a pre-processing step for the current MISes finder program. Moreover, the specification rules can be combined to stipulate system requirements. We are currently investigating how consistency validation can be composed/decomposed for such specification of combined rule bases.

Currently, our implementation uses SMTInterpol [8] as the underlying solver of the powerset manager and for checking satisfiability of the problem constraints. While this is sufficient for our purpose, it would be of our interests to investigate other SMT solvers in place of SMTInterpol. Another possible improvement for our implementation is to take advantage of the incremental checking and backtracking ability of SMT solvers (*i.e.*, using push/pop operations).

Parallelism has been considered for extracting a MUS [5,15]. In a similar fashion, the problem of enumerating MUSes and MISes can take advantage of parallel and/or distributed architectures. In particular, enumeration of MUSes can be parallelised and distributed to a cluster. The essential point to consider is how to correctly and effectively use the powerset manager. Our formal model suggests that having a parallel/distributed version of the program is possible.

Parallel/distributed version of the program is also a solution to another limitation of the current MISes finder program. Currently, our MISes finder program will terminate (without finding all MISes) if the underlying SMT solver cannot solve a satisfiability problem (*i.e.*, return *unknown*). This is often the case when the solver gets a "bad seed" such that its performance is deteriorated. By trying to solve several seeds at once, the MISes program can proceed even if some of the seeds are bad. Moreover, if the bad seeds are not MUSes or MSSes, the program can even terminate finding all MISes.

Often rule bases are developed step-by-step and subject to regular changes. It is necessary for the consistency validation to be carried out in an incremental fashion, where checks are only required to perform on the parts of the rule base that are affected by the changes. This is important for building a practical tool set supporting the development of specification rules.

The correctness of our MISes finder program relies on the separation between input and output constraints, in particular, the output constraints does not refer to any input variable. This is sufficient to model several regulations and policies. In the case where the output constraints refer to the input variables, our program does not guarantee to find all MISes for a rule base. What can be inferred is that any result of the program is an inconsistent set of rules (not necessarily minimal). Further investigation is required to validate this general type of rules, in particular, to consider solving constraints with quantifiers.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (2010)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
3. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 174–186. Springer, Heidelberg (2005)
4. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
5. Belov, A., Manthey, N., Marques-Silva, J.: Parallel MUS extraction. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 133–149. Springer, Heidelberg (2013)
6. Belov, A., Marques-Silva, J.: MUSer2: an efficient MUS extractor. JSAT **8**(1/2), 123–128 (2012)
7. Berstel, B., Leconte, M.: Using constraints to verify properties of rule programs. ICST **2010**, 349–354 (2010)
8. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012)
9. Hoang, T.S., Itoh, S., Oyama, K., Miyazaki, K., Kuruma, H., Sato, N.: Validating the consistency of specification rules. http://deploy-eprints.ecs.soton.ac.uk/465/ (2015)
10. Liffiton, M.H., Malik, A.: Enumerating infeasibility: finding multiple MUSes quickly. In: Gomes, C., Sellmann, M. (eds.) CPAIOR 2013. LNCS, vol. 7874, pp. 160–175. Springer, Heidelberg (2013)
11. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning **40**(1), 1–33 (2008)
12. Ligeza, A.: Logical Foundations for Rule-Based Systems. Studies in Computational Intelligence, vol. 11, 2nd edn. Springer, Heidelberg (2006)
13. Nadel, A., Ryvchin, V., Strichman, O.: Efficient MUS extraction with resolution. In: FMCAD 2013, pp. 197–200. IEEE (2013)
14. Berstel-Da Silva, B.: Verification of Business Rules Programs. Springer, Heidelberg (2014)
15. Wieringa, S.: Understanding, improving and parallelizing MUS finding using model rotation. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 672–687. Springer, Heidelberg (2012)