Validating the Consistency of Specification Rules*

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

Yokohama Research Laboratory, Hitachi Ltd.,
Kanagawa 244-0817, Japan,
hoang.thaison.ex, shinji.itoh.wn, kyohei.oyama.ec, kunihiko.miyazaki.zt, hironobu.kuruma.zg, naoto.sato.je@hitachi.com ,
http://www.hitachi.com/rd/yrl/index.html

Abstract. This paper focuses on the consistency analysis of specification rules, those expressing relationships between input and expected output of systems. We identify the link between Minimal Inconsistent Sets (MISes) of rules and Minimal Unsatisfiable Subsets (MUSes) of constraints. Furthermore, we develop a novel algorithm using SMT solvers for fast enumeration of MUSes, an essential component for practical validation of rules' consistency. We evaluate the algorithm using publicly available benchmarks. Finally, we apply our developed technique to check consistency of specifications rules of examples extracted from actual case studies.

Keywords: Specification rules, validation, Minimal Inconsistent Sets, Minimal Unsatisfiable Subsets, SMT

1 Introduction

Specification Rules. 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 some policies for a vehicle insurance company. Beside the normal contracts, the company offers two special rewards in the form of some *discount* (in percentages) for the insurance and some shopping *coupon*. The availability of the special rewards to customers depends on the *duration* (in number of years) of the contract, their online *account* status (whether or not they already have some 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 coupon of 100$ 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.

* Version: 09/01/2015

(R3) If the customer is not a VIP and the duration of the contract 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.

Each rule is made up of a constraint on the input and a constraint on the output. In particular, the rules are *non-deterministic*, *i.e.*, given a rule, there could be more than one possible output for a given input satisfying the rule. This type of "specification rules" are in particularly useful in the early designing process of the system where requirements are obscure and the details of the system cannot be decided at an early stage. Furthermore, the specification rules are gradually embodied. Non-determinism is essentially what makes *specification rules* different from *production rules* used in BRMS systems [7]. Production rules are designed for execution hence they are necessarily deterministic. More discussion on the similarities and differences between specification rules and production rules can be seen in Section 6.

The Rules Validation Problem. Given a set of rules, *i.e.*, the 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 possible input. Otherwise, the policies and regulations represented by the rule base are infeasible and cannot be implemented. In the example of the vehicle insurance company, the policies are inconsistent. More specifically, when a customer is a VIP with a 3-year contract, and has an online account, there are no possible values for the insurance discount and the shopping coupon satisfying the policies. Other properties of interests for a rule base are redundancy and completeness.

Motivation. This paper focuses on the *consistency analysis* of a special type of specification rules, namely those where the output constraints does not refer to the input (the vehicle insurance example is one of them). This type of specification rules is sufficient to stipulate many policies in financial and public sectors such as in taxation regulations or in the insurance industry. 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 consistency of any combination of the rules for validation. Our motivation is to develop some program/algorithms that can *efficiently* checking the validity of specification rules.

Current technologies. Within our knowledge, there are no existing technologies for checking consistency of specification rules. However, given the fact that each rule is made up of an input constraint and an output constraint, consistency of a rule base is related to the satisfiability of input and output 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) of an unsatisfiable constraint set [6, 11, 13]. Recently, algorithms for finding all MUSes have been proposed [3, 10, 9].

Approach. In order to validate the consistency of a set of specification rules, we enumerate all *Minimal Inconsistent Sets* (MISes) of the rule base. A MIS of the rule base is a set of inconsistent rules that is minimal, with respect to the set-inclusion ordering. Similar to constraints' MUSes, rules' MISes identify where the problems occur within the rule base. By exploring the relationship between the MISes of the rule base, MUSes of the output constraints of the rules, and satisfiability of individual input constraint, we reduce the problem of enumerating rules' MISes to that of output constraints' MUSes. In our approach, we use SMT solvers as black-boxes for solving satisfiability problems. Furthermore, our approach is constraint-agnostic, *i.e.*, independent of the types of the input/output constraints.

Contribution. Our first contribution is identifying the relationship between specification rules' MISes, output constraints' MUSes, and input constraints' satisfiability. Our second 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 [9] 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]. The detailed formal models can be found in Appendix A.

Structure. The rest of the paper is structured as follows. In Section 2, we present some background information for the work, including the problem of constraints satisfiability and rules consistency. In Section 3, we discuss the relationship between MISes and MUSes and show that the problem of finding MISes can be reduced to enumerating MUSes. In Section 4, we present a novel and efficient algorithm for enumerating MUSes. In Section 5, we present our empirical analysis of the new algorithm and its application in finding MISes. We discuss related work in Section 6. We conclude and propose some future work in Section 7.

2 Background

2.1 Constraints Satisfiability

In this paper, we often discuss the satisfiability problems related to different generic sets of constraints. For each sets 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 $\mathbf{A}$ to the problem's variables that meets its restriction. We use the notation $\mathsf{sat}(\mathbf{A}, C)$ to denote the fact that $\mathbf{A}$ satisfies constraint C, and $\mathsf{unsat}(\mathbf{A}, C)$ otherwise.

Given a set of constraints $Cs \subseteq \mathbb{C}$, if there exists some assignment that satisfies every constraints in Cs then it is said to be satisfiable (SAT). Otherwise, Cs is infeasible or 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 subsets of a constraint set $\mathbb{C}$, namely: *Maximal Satisfiable (sub-)Set* (MSS) and *Minimal Unsatisfiable (sub-)Set* (MUS). A set of constraints $Cs \subseteq \mathbb{C}$ is a MSS if it is a satisfiable subset of $\mathbb{C}$ and cannot be expanded without compromising satisfiability, *i.e.*,

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

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

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

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

2.2 Rules Consistency

We focus on this paper on a generic sets 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 the constraints are not specified.

A rule $R_i = (I_i, O_i)$ is satisfied by an assignment $\mathbf{A}_x$ to the input variables and an assignment $\mathbf{A}_y$ to the output variable —denoted as $\mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R_i)$— if either the input assignment $\mathbf{A}_x$ *does not satisfy the input constraint* I_i or the output assignment $\mathbf{A}_y$ *satisfies the output constraint* O_i.

Definition 1 (Rule Satisfiability). *Given a rule R with an input constraint I and an output constraint O, an input assignment $\mathbf{A}_x$, and an output assignment $\mathbf{A}_y$, we define*

$$\mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R) \ \widehat{=} \ \mathsf{unsat}(\mathbf{A}_x, I) \ \vee \ \mathsf{sat}(\mathbf{A}_y, O) \ . \tag{5}$$

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

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

$$\mathsf{Consistent}(S) \;\widehat{=}\; \forall \mathbf{A}_x \cdot \exists \mathbf{A}_y \cdot \forall i \in S \cdot \mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R_i) \;, \text{ and} \tag{6}$$

$$\mathsf{Inconsistent}(S) \;\widehat{=}\; \exists \mathbf{A}_x \cdot \forall \mathbf{A}_y \cdot \exists i \in S \cdot \neg \mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R_i) \;. \tag{7}$$

From now on, we will use *set of rules* for *set of rule indices*, when there is no ambiguity.

Given a rule base $\mathbb{R}$, our interests are to validate if the rule base is consistent. Moreover, in the case where it is inconsistent, some indicating facts about the rule base should be given to "explain" the inconsistency. Since consistency is anti-monotonic with respect to the set-inclusion ordering of the rules, we define the following notion of *Minimal Inconsistent Set* (MIS).

Definition 3 (MIS). *Given a set of rules $S \subseteq 1 \mathinner{..} n$, S is a MIS if and only if S is inconsistent and is minimal with respect to the set-inclusion ordering.*

$$\mathsf{MIS}(S) \;\widehat{=}\; \mathsf{Inconsistent}(S) \;\wedge\; (\forall T \cdot T \subset S \Rightarrow \mathsf{Consistent}(T)) \tag{8}$$

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. Similar to the role of MUSes with respect to constraints satisfiability, the MISes of $\mathbb{R}$ are the "inconsistent core" of $\mathbb{R}$. In general, the problem of finding all MISes is intractable: the number of MISes may be exponential in the size of the rule base. The main objective of our work here is to *quickly enumerate MISes*.

Example 1. Consider the vehicle insurance company's policies mentioned in Section 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	$discount \geq 5 \wedge coupon \in 50 \mathinner{..} 100$
R_3	$\neg VIP \wedge duration < 2$	$discount < 5 \vee coupon \in 30 \mathinner{..} 50$
R_4	$VIP \wedge duration \geq 2$	$discount \geq 7 \wedge coupon = 50$

In the above example, input assignment assigning FALSE to $account$, TRUE to VIP, 1 to $duration$ and output assignment assigning 5 to $discount$, 50 to $coupon$ satisfy the rules R_1–R_4. The set of rules is inconsistent (as mentioned before) and has one MIS which is the set of rules $\{R_1, R_4\}$.

(End of example)

3 Relationship between MISes and MUSes

In this section, we investigate the relationship between the MISes and satisfiability problems on the input and output constraints. Essentially, we look for an approach to solve the rule consistency problem by solving several satisfiability problems. Since the input and output variables are disjoint, satisfibility problems on input constraints and output constraints are independent of each other. Given a set of rules S, we first consider the consistency of S, *i.e.*, $\mathsf{Consistent}(S)$, knowing the answers for the satifiability of its input and output constraints (*e.g.*, $\mathsf{SAT}(I[S])$ or $\mathsf{UNSAT}(I[S])$, $\mathsf{SAT}(O[S])$ or $\mathsf{UNSAT}(O[S])$). Here, we use the notation $C[S]$ to denote the set of constraints C_i where i is in S, *i.e.*, $C[S] = \{C_i \mid i \in S\}$.

Lemma 1. *Given a set of rule S, we have*

$$\mathsf{SAT}(I[S]) \wedge \mathsf{UNSAT}(O[S]) \quad \Rightarrow \quad \mathsf{Inconsistent}(S)\ . \tag{9}$$

Proof (Sketch). The proof is straight-forward by expanding Definition 2 for $\mathsf{Inconsistent}$, Definition 1 for rsat, together with (1) and (2).

Notice that (9) is an implication rather than an equivalence. In particular, $\mathsf{UNSAT}(I[S]) \Rightarrow \mathsf{Consistent}(S)$ does not hold. However, we still have the following relationship between $\mathsf{SAT}(O[S])$ and $\mathsf{Consistent}(S)$.

Lemma 2. *Given a set of rule S, we have*

$$\mathsf{SAT}(O[S]) \quad \Rightarrow \quad \mathsf{Consistent}(S)\ . \tag{10}$$

Proof (Sketch). The proof is straight-forward by expanding Definition 2 for $\mathsf{Consistent}$, Definition 1 for rsat, together with (1).

The relationship between rule consistency and input/output constraints satisfiability is summarised in Table 1. The *unknown* entry in Table 1 indicates

$\mathsf{SAT}(I[S])$	$\mathsf{SAT}(O[S])$	$\mathsf{Consistent}(S)$
T	T	T
T	F	F
F	T	T
F	F	*unknown*

Table 1: Consistency and Satisfiability

that we cannot determine consistency of a set of rules just by checking satisfibility of its input and output constraints. In particular, in the case where $\mathsf{UNSAT}(I[S])$ and $\mathsf{UNSAT}(O[S])$, consistency of S is determined by the consistency of its proper-subsets.

Example 2. Consider rules R_1–R_4 from Example 1. The following subsets of rules have their input and output constraints unsatisfiable, but their consistency statuses are different.

- $\{R_1, R_2, R_3, R_4\}$: Inconsistent
- $\{R_1, R_2, R_3\}$: Consistent
- $\{R_1, R_3, R_4\}$: Inconsistent

(End of example)

As a result, a naïve way for enumerating the MISes of a rule set is by iterating through its subsets from lower to higher cardinality, as the following naiveMISesFinder algorithm.

```
   naiveMISesFinder  ≙
   output: MISes as they are found
1. MISes := ∅;
2. for i from 1 to n
3.    foreach S ⊆ 1 .. n ∧ card(S) = i ∧ (∀mis ∈ MISes·mis ⊈ S)
4.       if SAT(I[S]) ∧ UNSAT(O[S])
5.          yields S;
6.          MISes := MISes ∪ {S};
```

The condition at Line 3 ensures that we consider only those set of rules S which is not a super-set of any MIS that are already found. Since the algorithm iterates the subset of rules from lower to higher cardinality, at Line 4, all proper subsets of S (who cardinality are smaller than $card(S)$) are consistent. Subsequently, checking for satifiability of input/output constraints of S is sufficient to determine its consistency. Furthermore, if S is inconsistent, it is also minimal, *i.e.*, a MIS. However, the above algorithm is rather too inefficient for checking rules consistency. In particular, in the case where the rules are consistent, naiveMISesFinder needs to enumerate all possible subsets of the set of rules.

To avoid iterating the subsets of rules, we prove the following relationship between MISes, MUSes of the output constraints, and satisfiability of input constraints.

Theorem 1 (The MISes and the MUSes). *A subset of rule S is a MIS iff S's output constraints are a MUS and S's input constraints are satisfiable.*

$$\mathsf{MIS}(S) \quad \Leftrightarrow \quad \mathsf{MUS}(O[S]) \wedge \mathsf{SAT}(I[S]) \ . \tag{11}$$

Proof. We prove (11) in each direction separately.

1. *From left to right*: Let S be a set of rules such that MIS(S), we prove that SAT($I[S]$) (see (1a)) and MUS($O[S]$) (see (1b)). First, since MIS(S), from Definition 3, S is inconsistent, *i.e.*, Inconsistent(S). From Definition 2 for Inconsistent, there exists an input assignment $\mathbf{A}_x$ such that

$$\forall \mathbf{A}_y \cdot \exists i \in S \cdot \neg \mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R_i) \ . \tag{12}$$

(a) We continue to prove $\mathsf{SAT}(I[S])$ by contradiction. Assuming $\mathsf{UNSAT}(I[S])$, instantiate (1) with $\mathbf{A}_x$, there exists a rule index $i \in S$ such that

$$\mathsf{unsat}(\mathbf{A}_x, I_i) \tag{13}$$

Since $\mathsf{MIS}(S)$, from Definition 3, $S \setminus \{i\}$ is consistent, *i.e.*, $\mathsf{Consistent}(S \setminus \{i\})$. As a result, instantiate (6) in Definition 2 with $\mathbf{A}_x$, there exists an output assignment $\mathbf{A}_y$ such that

$$\forall k \in S \setminus \{i\} \cdot \mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R_k) \tag{14}$$

Instantiate (12) with $\mathbf{A}_y$, there exists a rule index j such that

$$\neg\mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R_j)\ . \tag{15}$$

From Definition 1 and (15), we have $\mathsf{sat}(\mathbf{A}_x, I_j)$. Together with (13), we have $i \neq j$. Instantiate k in (14) with j, we obtain $\mathsf{rsat}((\mathbf{A}_x, \mathbf{A}_y), R_j)$, which is in contradiction with (15).

(b) In order to prove $\mathsf{MUS}(O[S])$, according to (4), we prove that $\mathsf{UNSAT}(O[S])$ and $\forall T \cdot T \subset S \Rightarrow \mathsf{SAT}(O[T])$.

i. From (12) and Definition 1, we have

$$\forall \mathbf{A}_y \cdot \exists i \in S \cdot \mathsf{sat}(\mathbf{A}_x, I_i) \wedge \mathsf{unsat}(\mathbf{A}_y, O_i) \tag{16}$$

According to (2), we have $\mathsf{UNSAT}(O[S])$.

ii. We now prove that $\mathsf{SAT}(O[T])$ for any $T \subset S$. Since $\mathsf{MIS}(S)$, from Definition 3, T is consistent, *i.e.*, $\mathsf{Consistent}(T)$. We continue the proof by constradiction, *i.e.*, to assume $\mathsf{UNSAT}(O[T])$. Since $\mathsf{SAT}(I[S])$ (from (1a)), we have $\mathsf{SAT}(I[T])$. Applying Lemma 1, we obtain $\mathsf{Inconsistent}(T)$ which is a contradiction.

2. *From right to left*: Let S be the set of rules such that $\mathsf{SAT}(I[S])$ and $\mathsf{MUS}(O[S])$, we prove that $\mathsf{MIS}(S)$. According to Definition 3, we have to prove that $\mathsf{Inconsistent}(S)$ and $\forall T \cdot T \subset S \Rightarrow \mathsf{Consistent}(T)$.

(a) From $\mathsf{MUS}(O[S])$, we have $\mathsf{UNSAT}[O[S]]$. Together with $\mathsf{SAT}(I[S])$, we have $\mathsf{Inconsistent}(S)$ (Lemma 1).

(b) For any $T \subset S$, we have $\mathsf{SAT}(O[T])$. As a result, we obtain $\mathsf{Consistent}(T)$ (Lemma 2).

□

Theorem 1 reduces the problem of enumerating MISes to finding output constraints MUSes and checking the satisfiability of the input constraints of 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 efficient algorithm for quickly enumerating MUSes.

4 An Efficient Algorithm for Enumerating MUSes

While there are many algorithms for extracting a single MUS from an unsatisfiable set of constraints, there are only a few programs for enumerating MUSes. Our algorithm for enumerating MUSes is inspired by the state-of-the-art algorithm MARCO [9]. The main feature of MARCO is the use of a powerset manager maintaining a powerset map for selecting unexplored subsets. We first review the original MARCO algorithm in Section 4.1 before presenting our algorithm called MUSesHunter in Section 4.2.

4.1 The MARCO algorithm

In order to find the MUSes of a constraint set $\mathbb{C}$ without enumerating through the subsets of constraints, the key novelty of the MARCO is to maintain a powerset map, a propositionnal formula keeping track of the "unexplored subset". A SAT solver is used to solve the map to get an unexplored subset of constraints. The map is effectively pruned during the execution of the algorithm. In the subsequent, we give details of a powerset manager maintaining the powerset map.

The powerset manager. The powerset manager maintains 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 $\mathbb{C}$. The set of propositions Ps corresponds to the set of unexplored subsets. There are three basic operations for the powerset manager, namely getSet, addLowerBound and addUpperBound as showed in Figure 1. In

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

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

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

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

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

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

Fig. 1: Basic operations of the powerset manager

getSet, the powerset utilises the capability of a SAT solver to return a model for a

set of satisfiable constraints. In particular, a model for satisfiable Ps corresponds to an unexplored subset. Operations addLowerBound and addUpperBound are for restricting Ps. Operation addLowerBound(L) marks all subsets of the input set L as explored. Similarly addUpperBound(U) marks all supersets of the input set U as explored.

MARCO. The MARCO algorithm which makes use of the powerset manager can be seen in Figure 2. Intuitively, for each iteration, MARCO gets a new unexplored subset S from the powerset manager. If the set of constraints corresponding to S, *i.e.*, $\mathbb{C}[S]$, is satisfiable, MARCO uses a grow sub-routine to obtain an MSS, and adds this MSS as a lower bound to restrict future iterations. Otherwise, if UNSAT($\mathbb{C}[S]$), MARCO uses a shrink sub-routine 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 there is no MUS which is 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). Figure 3 illustrates some possible implementations for grow and shrink sub-routines. Operation growLin gradually adds new elements to a satisfiable subset S if it 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 example, the shrinkBin and its sub-routine reduce perform binary search and can return a MUS faster than shrinkLin. A similar binary search algorithm also exists for the grow routine. The MARCO algorithm and various grow and shrink routines are not novel.

```
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: Different implementations of grow and shrink routines

Example 3. An example execution trace for MARCO (applying to the output constraints of the rules R_1–R_4 in Example 1) is showed in Table 2. At each step, we report the number of SMT calls (*i.e.*, for checking satisfiability of the problem constraints) and the number of SAT calls (*i.e.*, for making querries to the powerset manager). The MARCO program uses growLin and shrinkBin subroutines for growing and shrinking the seeds accordingly. In total, MARCO found 2 MUSes and 3 MSSes using 21 SMT calls and 5 SAT calls.

Step		Seed (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

Table 2: Example trace of MARCO algorithm

(End of example)

4.2 The MUSesHunter Algorithm

Our observation of the MARCO algorithm in Figure 2 is that the role of the powerset manager only used for retrieving unexplored subset of constraints. In particular, during the process of growing and shrinking, many checks for satifiability of subsets are spurious: these subsets are either supersets of some MUSes or subsets of some MSSes. This is particularly expensive when the type of constraints requires the solver to deal with theory other than just Boolean constraints. The powerset manager only deals with satisfiability of Boolean constraints, hence potentially less expensive. As a result, we use the powerset manager for checking satisfiability of the problem constraints in $\mathbb{C}$ whenever possible.

Another observation is that during the process of growing, often unsatisfiable subsets of $\mathbb{C}$ are found. If we call shrink sub-routine on these unsatisfiable subsets, we can get MUSes faster. The challenge here is to ensure that by shrinking immediately follows some grow steps, we obtain a *new* MUS. Once again, we make use of the powerset manager for that purpose.

The new algorithm (called "MUSesHunter") can be seen in Figure 4. Compare to MARCO, the main difference is the use of the powerset manager within the

```
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 satifiable
4.      set ← growHyb(S);            //     grow (hybrid) 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 an 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 unsatifiable
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 an lower bound.
16.   S ← getSet();                  //   Get a new unexplored subset S
```

Fig. 4: MUSesHunter algorithm

growHyb and shrinkBinPS sub-routines. In particular, growHyb returns either a MUS or an MSS. As a result, MUSesHunter can enumerate MUSes faster than MARCO. Furthermore, in the case where a MUS is found, we add that MUS both as a lower bound and an upper bound. In the MARCO algorithm, a MUS is only added to the powerset manager as an upper bound. As a result, our MUSesHunter restricts the subsets of constraints faster than MARCO. In the subsequent, we discuss in details how to use the powerset manager for satifiability checking and to obtain a new MUS by first growing to some unsatifiable set then shrinking.

Using Powerset Manager for Satifiability Checking In order to use the powerset manager for satifiability checking of subsets of constraints, a new operation isUnexplored(S) for checking if S is unexplored is added as showed in Figure 5. This can be done by a satifiability checking of the current set of constraints Ps representing the unexplored subsets together with the constraint representing the input set S.

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

Fig. 5: The isUnexplored routine

The following theorem states an important property of the powerset manager in the MUSesHunter algorithm, in particular, of the explored subsets as filtered out by the powerset manager.

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

$$\neg\mathsf{isUnexplored}(S) \quad \Leftrightarrow \quad \begin{array}{l} (\exists L \cdot \mathsf{SAT}(\mathbb{C}[L]) \wedge S \subseteq L) \\ \vee\ (\exists M \cdot \mathsf{MUS}(\mathbb{C}[M]) \wedge S \subset M) \\ \vee\ (\exists M \cdot \mathsf{MUS}(\mathbb{C}[M]) \wedge M \subseteq S)\ . \end{array} \tag{17}$$

Proof. This theorem states that a subset of constraints S is explored by the powerset manager if either (1) there exists an satisfiable L which is a superset of S, or (2) there exists a MUS which is a (proper-)superset of S, or (3) there exists a MUS which is a subset of S. This holds trivially (as an invariant) for the MUSesHunter algorithm, since

- when an MSS is found, it is added as a lower bound for unexplored subsets, and
- when a MUS is found, it is added as a lower bound and upper bound for unexplored subsets.

□

The following Lemmas are consequences of Theorem 2.

Lemma 3 (Satifiability during shrink). *Given sets of constraints* S *and* T*, if we have*

1. $\mathsf{isUnexplored}(S)$,
2. $T \subseteq S$,
3. $\neg\mathsf{isUnexplored}(T)$,

then $\mathsf{SAT}(\mathbb{C}[T])$.

Proof. From Condition 3, apply Theorem 2, we have three cases as follows.

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

Lemma 4 (Unsatifiability during grow). *Given sets of constraints* S *and* T*, if we have*

1. $\mathsf{isUnexplored}(S)$,
2. $S \subseteq T$,
3. $\neg\mathsf{isUnexplored}(T)$,

```
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 unsatifiable,
 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 unsatifiable,
 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
```

Fig. 6: The shrink routine with powerset manager

then UNSAT(ℂ[T]).

Proof. The proof is similar to the proof of Lemma 3 and is omitted here.

Lemma 3 and Lemma 4 allow us to use the powerset manager to replace some of the satifiability checking for subsets of ℂ during shrinking and growing sub-routines. In particular, Lemma 3 ensures the correctness of shrinkBinPS showed in Figure 6. Compare to the reduce routine in Figure 3, in reducePS, before checking satifiability of $C \cup B$ (Line 3) and $D \cup B$ (Line 6), we first check if these subsets are unexplored. Lemma 3 ensures that they are already explored, they are satisfiable.

The growHyb sub-routine The growHyb is showed in Figure 7 returns either a new MSS or a new MUS. It is based on the growLin routine showed earlier in Figure 3. Similar to reducePS routine, before checking satifiability for $S \cup \{c\}$ (Line 3), the growHyb routine check if $S \cup \{c\}$ is unexplored. Lemma 4 ensures that if $S \cup \{c\}$ is already explored, it is unsatifiable. Moreover, the fact that $S \cup \{c\}$ in Line 6 is unexplored guaranteed that calling shrinkBinPS on $S \cup \{c\}$ will return a new MUS.

Example 4. An example of an execution trace for the MUSesHunter algorithm applying to the set of output constraints for the rules R_1–R_4 in Example 1 is show in Table 3. The MUSesHunter program use growHyb and shrinkBinPS for

```
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 satifiable,
4.       S := S ∪ {c};                    //       add c to S
5.     else                               //     if S ∪ {c} is unsatifiable
6.       return shrinkBinPS(S ∪ {c});     //       shrink to find a MUS
7. return S;                              // return S which is an MSS
```

Fig. 7: The growHyb routine

growing and shrinking the seeds accordingly. In total, MUSesHunter found 2 MUSes and 1 MSSes using 9 SMT calls and 15 SAT calls.

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

Table 3: Example trace of MUSesHunter algorithm

(End of example)

5 Empirical Analysis

We implement our algorithm for finding MUSes and MISes using Java. In particular, for constraint solving (both for querying the powerset manager and checking satisfiability of the problem constraints), we use SMTInterpol [8]. For analysis, we first compare the performance of MUSesHunter and MARCO algorithms. Afterwards, we evaluate the performance of developed MISes finder program using MUSesHunter algorithm against examples of various sizes.

5.1 MUSesHunter vs. MARCO

Both MUSesHunter and MARCO algorithms were implemented using the same underlying infra-structured, sharing as much code as possible. In particular, they use identical solvers for the powerset manager and problem constraints satisfiability checking. For growing and shrinking, MARCO uses growLin and shrinkBin sub-routines, whereas MUSesHunter uses growHyb and shrinkBinPS sub-routines. However, the powerset manager is 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 4x2.7GHz CPUs running Linux. Each program is running with 3GB heap memory limit and an 1800-second timeout. There is no timeout for individual constraints satisfiability checking. We selected 473 samples selected from SMT-LIB for quantifier-free linear integer arithmetic (QF_LIA). The benchmarks were of different sizes ranging from 4 to 881 constraints[1].

	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	473		473	
Max MUSes found per sample	29740		18646	
Average MUSes Found	1050		533	
Total satisfiability calls	7749472		3127773	
SATs		6472339 (84%)		82628(3%)
SMTs		1277133 (16%)		3045145(97%)

Table 4: Summary

Overall. The summary of the results for running the two algorithms is presented in Table 4. While the numbers of cases where the algorithms terminate and find all MUSes are comparable, MUSesHunter tends to run out of memory whereas MARCO tends to run out of time 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 examples (105 cases), whereas that percentage for MUSesHunter is 6% (21 cases). This is the direct effect of the growHyb sub-routine used in the MUSesHunter algorithm: 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.

[1] Available from the `root/SMT/SMT-LIB benchmarks/2014-06-03/` space at `https://www.starexec.org/`. Conversion to the expected input format of the MUSes finder programs are required.

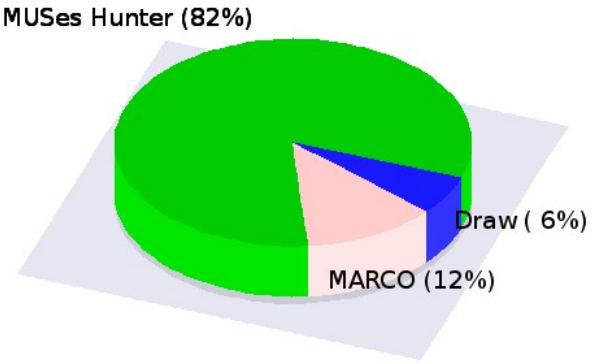

Fig. 8: MUSesHunter vs. MARCO performance

Table 5 reports on the direct performance comparison between MUSesHunter and MARCO algorithms on the selected examples. 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, *i.e.*, either algorithm runs out of time or memory, we compare the number of MUSes that the algorithms found.

	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%)

Table 5: Performanace comparison between MUSesHunter and MARCO

Overall, MUSesHunter outperforms MARCO by finding more MUSes and in shorter time (82%). In particular, most of the case where MARCO outperforms MUSesHunter, it is due to the fact that MUSesHunter is either timeout or out-of-memory. This suggests that with a perfect oracle (*i.e.*, one which can solve all combinations of constraints within a reasonable time), MUSesHunter will perform better than MARCO.

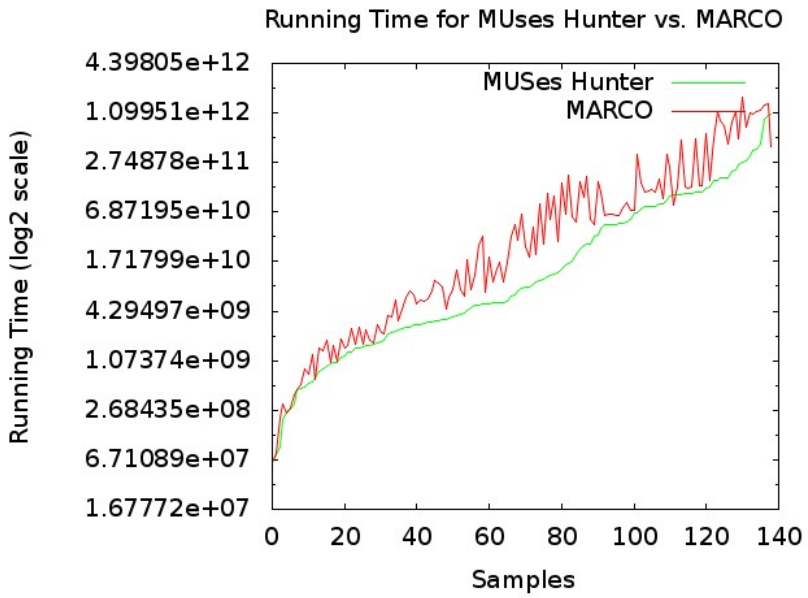

Fig. 9: MUSesHunter vs. MARCO running time

Both programs terminate. The comparison between the (log-scale) speed of MUSesHunter and MARCO algorithm in the case where they both terminate can be seen in Figure 9. In most cases (97%), MUSesHunter terminates faster than MARCO. In particular, 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 heavily makes use of the powerset manager as a substitute for checking satisfiability of the problem constraints whenever possible. Even though the total number of checking for satisfiability for MUSesHunter is twice as many as that of MARCO, checking satisfiability for the powerset manager (solving Boolean constraints satisfiability problems) is much easier and faster than checking satisfiability of the problem constraints (solving satisfiability of quantified-free linear integer arithmetic constraints). The percentage constraint solver usage for MUSesHunter and MARCO can be seen in Figure 10.

Moreover, when a MUS is found, MUSesHunter blocks both its super-sets as well as subsets, where MARCO only blocks the MUS' super-sets. Together with the preferences of finding MUSes over MSSes in the growing sub-routines, MUSesHunter prunes the search space much faster than MARCO. Compare the traces for MARCO and MUSesHunter in Table 2 and Table 3. MUSesHunter does not need to find all MSSes before terminate. 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.

(a) MUSesHunter (b) MARCO

Fig. 10: MUSesHunter vs. MARCO. Constraint solver usages

Figure 11 compares the percentage of MUSes found against the time, both scaled to the range $0..1$ for examples 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, the MUSes are found gradually.

One program does not terminte. In the case where one of the algorithm does not terminate (either out of time or memory), we focus on the number of MUSes found by each algorithm. In 75% cases, MUSesHunter found more MUSes than MARCO. Figure 12 shows the comparison between the numbers of MUSes found by MUSesHunter and MARCO.

5.2 MISes Finder

We use the MISes finder program on examples extracted from some industrial case studies in financial and public sectors. The performance of the MISes finder

Example	Size	Time	MUSes	MSSes	MISes	SATs	SMTs
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	1800s	436	1093	0	820455	373767

Table 6: MISes finder performance

program largely depend on the underlying MUSesHunter algorithm on finding MUSes of the output constraints. The program is evaluated with example of

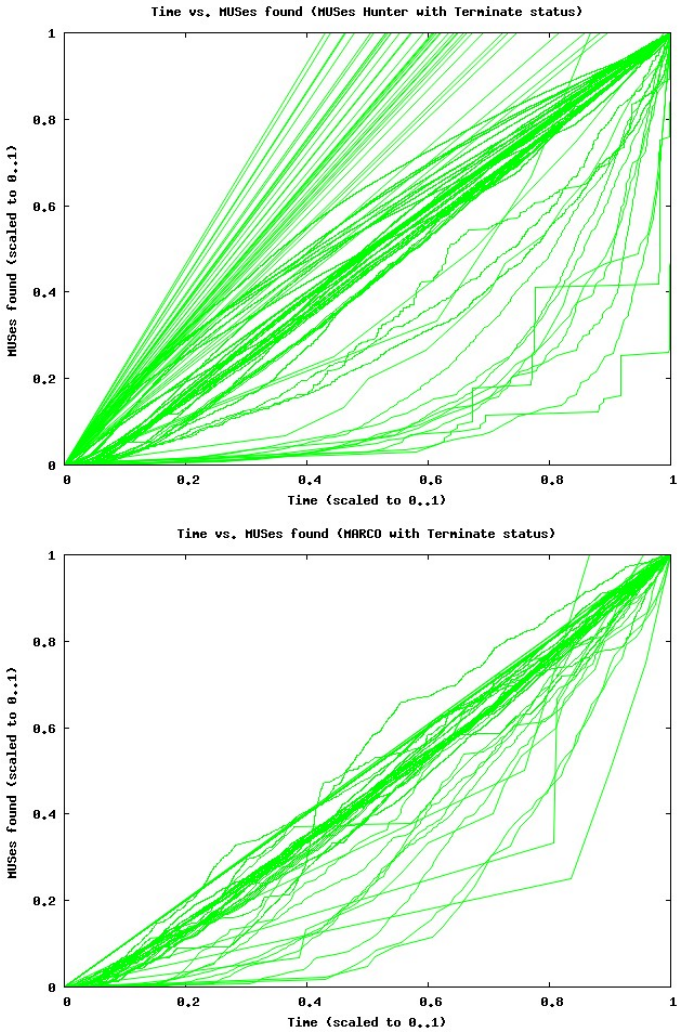

Fig. 11: MUSes found vs. Time

different size. The last two examples (namely *Vehicle tax* and *Registration*) are extracted from actual policies and are indeed consistent. MISes finder program does not terminate for the largest example of 725 rules (*Registration*). For this example, all 436 MUSes (but none of them are MISes) are quickly found within 60 seconds. Afterwards, the program only found MSSes. Given the size of the rule, the number of MSSes need to be found are large we do not expect the program to terminate within reasonable time. To validate this set of rules, we need to adopt some additional techniques to reduce the complexity of the problem.

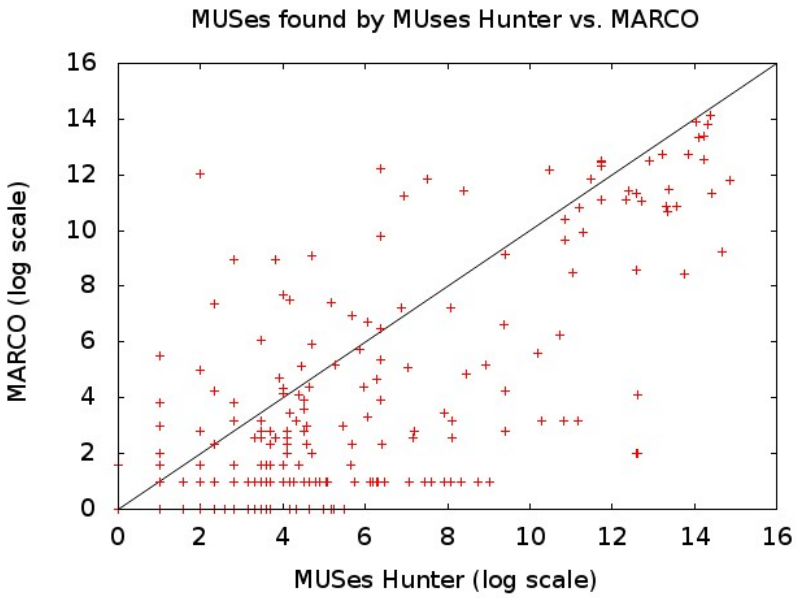

Fig. 12: Number of MUSes found comparison

6 Related Work

Our MISes finder program is an essential part of a suite for developing regulations and policies using specification rules. Each production rule is also composed of a guard (input constraints) and an action (output constraints). The main difference between specification rules and production rules used in BRMS systems [7] is the fact that specification rules can be non-deterministic while production rules are deterministic. Furthermore, the purpose of specification rules is to constraint the relationships between input and output of the system without referencing the system state. The production rules often involve system with explicit state (its working memory). Production rules are also written with some implicit rule execution engine in mind, *e.g.*, firing enabled rules repeatedly while evolving the state of the system.

In term of rules validation, similar properties have been investigated [12] production rules. For consistency, due to its deterministic nature, minimal inconsistent for production rules can only be between two rules, *i.e.*, pairwise. Other properties for production rules that have been considered include *redundancy* and *completeness*. Similar concepts can be associated with specification rules and are a part of our future research.

Regarding the problem of enumerating MUSes, our algorithm is inspired by the MARCO algorithm to avoid iterating the subset of constraints by using a powerset manager. The novelty of our approach is the use of the powerset manager for reducing the number of satisfiability checks of the problem constraints

and prioritise finding MUSes over MSSes. Essentially, we replace these expensive satisfiability checks of problem constraints with satisfiability checks of Boolean constraints in the powerset manager. The performance gain is in proportion to the ratio of difficulty between solving the problem constraints, *e.g.*, quantifier-free linear integer arithmetic (QF_LIA), and Boolean constraints.

In order to find all MISes of a set of rules, we need to find all MUSes of its output constraints. Comparison in [9] suggests the CAMUS algorithm [10] 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 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 quicker can be of useful to validate a consistent set of rules. We plan to investigate and evaluate CAMUS further.

7 Conclusion

In this paper, we present our approach for validating consistency of specification rules, those describe the relationship between the system input and expected output. Our method explores the relationship between *Minimal Inconsistent Sets* (MISes) of rules, *Minimal Unsatisfiable (sub-)Sets* (MUSes) of output constraints, and satisfiability of input constraints. We developed a novel algorithm for quick enumeration of MUSes during the validation process and evaluated it against MARCO [9], 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 well-known problem such as shrink and grow sub-routines to find MUSes and *Maximal Satisfiable (sub-)Sets* (MSSes). Any state-of-the-art implementation for these sub-routines can be plugged in our approach.

As mentioned before, 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 set of rules. Rule separation will drastically reduce the complexity of the MISes finding problem, hence could be used as the pre-processing step for the current MISes finder program. Moreover, the specification rules can be combined to stipulate systems' requirements. We are currently investigating how consistency validation can be composed/decomposed for such specification of combined rule bases.

Parallelism has been considered for extracting a MUS [5, 13]. In a similar fashion, the problem of enumerating MUSes and MISes can take advantage of parallel and/or distributed architecture. In particular, the search for MUSes can be parallelised and distributed to a cluster. The key important point to consider is how to correctly and effectively use a shared 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 problem with the current MISes finder program. Currently, our MISes finder program will terminate (without finishing finding all MISes) if the underlying SMT solver cannot solve the any constraint satisfiability problem (*i.e.*, return *unknown*). This is often the case where the constraint solver gets a "bad seed" that its performance is deteriorated. By trying to solve several seeds at once, the MISes program can make progress even if some of the seeds are bad. Moreover, if the bad seeds are not MUSes or MSSes, the program can even terminate and find all MISes.

Often the rule bases are developed step-by-step and changes are made regularly. It is necessary for rule validation program to be design such that it can perform in an incremental fashion, where checks only needs to be done for the parts of the rule base that are effect by the changes. This is important for any practical application to build a tool set for 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 for us to apply 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 inconsistency for a rule base. What can be inferred is that any found MIS is an inconsistent set of rules (but not necessarily minimal). Further investigation is required to validate this general type of rules, in particular, in solving constraints with quantifiers.

References

1. J-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, November 2010.
3. James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005*, volume 3350 of *Lecture Notes in Computer Science*, pages 174–186, Long Beach, CA, USA, January 2005. Springer. `http://dx.doi.org/10.1007/978-3-540-30557-6_14`.
4. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter Satisfiability Modulo Theories, pages 825–885. IOS Press, 2009. `http://dx.doi.org/10.3233/978-1-58603-929-5-825`.
5. Anton Belov, Norbert Manthey, and João Marques-Silva. Parallel MUS extraction. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference*, volume 7962 of *Lecture Notes in Computer Science*, pages 133–149, Helsinki, Finland, July 2013. Springer. `http://dx.doi.org/10.1007/978-3-642-39071-5_11`.

6. Anton Belov and João Marques-Silva. MUSer2: An efficient MUS extractor. *JSAT*, 8(1/2):123–128, 2012. http://jsat.ewi.tudelft.nl/content/volume8/JSAT8_9_Belov.pdf.
7. Bruno Berstel and Michel Leconte. Using constraints to verify properties of rule programs. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010*, pages 349–354, 2010. http://dx.doi.org/10.1109/ICSTW.2010.42.
8. Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In Alastair F. Donaldson and David Parker, editors, *Model Checking Software - 19th International Workshop, SPIN 2012*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254, Oxford, UK, July 2012. Springer. http://dx.doi.org/10.1007/978-3-642-31759-0_19.
9. Mark H. Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple MUSes quickly. In Carla P. Gomes and Meinolf Sellmann, editors, *CPAIOR 2013*, volume 7874 of *Lecture Notes in Computer Science*, pages 160–175, Yorktown Heights, NY, USA, May 2013. Springer. http://dx.doi.org/10.1007/978-3-642-38171-3_11.
10. Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008. http://dx.doi.org/10.1007/s10817-007-9084-z.
11. Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS extraction with resolution. In *Formal Methods in Computer-Aided Design, FMCAD 2013*, pages 197–200, Portland, OR, USA, October 2013. IEEE. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679410.
12. Bruno Berstel-Da Silva. *Verification of Business Rules Programs*. Springer, 2014. http://dx.doi.org/10.1007/978-3-642-40038-4.
13. Siert Wieringa. Understanding, improving and parallelizing MUS finding using model rotation. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012*, volume 7514 of *Lecture Notes in Computer Science*, pages 672–687, Québec City, QC, Canada, 2012. Springer. http://dx.doi.org/10.1007/978-3-642-33558-7_49.

A Event-B Formalisation of the Algorithms

An Event-B Specification of c0_0-iSTATE
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c0_0-iSTATE
This context declares the input states
SETS
$iSTATE$ The input states
END

An Event-B Specification of c0_0-oSTATE
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c0_0-oSTATE
This context declares the output states
SETS
$oSTATE$ The output states
END

An Event-B Specification of c0_1-iConstraints
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c0_1-iConstraints

This context declares an array of input constraints

EXTENDS c0_0-iSTATE
CONSTANTS

iConstraints The array of input constraints
AXIOMS

iConstraints-type : $iConstraints \in 1 \mathinner{..} size \rightarrow \mathbb{P}(iSTATE)$

Input constraints are numbered from 1 to size
Each input constraint corresponds to a set
of input states satisfying the constraint
In particular, $sat(a, C)$, i.e.,
input assignment a satisfying constraint C,
is modelled as $i \in C$.
Similarly $unsat(a, C)$ is modelled as $i \notin C$.

END

An Event-B Specification of c0_1-oConstraints
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c0_1-oConstraints

This context declares an array of output constraints
EXTENDS c0_0-oSTATE
CONSTANTS

oConstraints The array of output constraints
AXIOMS

oConstraints-type : $oConstraints \in 1 \mathinner{..} size \rightarrow \mathbb{P}(oSTATE)$

The output constraints are numbered from 1 to size.

Each output constraint corresponds to a set of output states satisfying the constraint.

In particular, sat(a, C), i.e., output assignment a satisfying constraint C,

is modelled as $i \in C$. Similarly unsat(a, C) is modelled as $i \notin C$.

END

An Event-B Specification of c0_2-rsat
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c0_2-rsat

This context defines rules' satisfiability by combining the corresponding input/output constraints' satisfiability

EXTENDS c0_1-iConstraints

CONSTANTS

$rsat$

AXIOMS

rsat-def : $rsat = (\lambda i \cdot i \in 1 \mathinner{..} size \mid \{in \mapsto out | in \notin iConstraints(i) \lor out \in oConstraints(i)\})$

A pair (@in, @out) satisfies a rule if
either @in does not satisfy the input constraint or
@out satisfies the output constraint.

rsat-type : $rsat \in 1 \mathinner{..} size \to \mathbb{P}(iSTATE \times oSTATE)$

Typing information

END

An Event-B Specification of c0_3-MISes
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c0_3-MISes

This context defines consistency for any set of rules and MISes

EXTENDS c0_2-rsat

CONSTANTS

$Consistent$ Consistency

$MISes$ Minimal Inconsistent Sets

AXIOMS

Consistent-def : $Consistent = \{rs | rs \subseteq 1 \mathinner{..} size \land (\forall in \cdot \exists out \cdot \forall i \cdot i \in rs \Rightarrow in \mapsto out \in rsat(i))\}$

A set of rules @rs is consistent iff
for all input @in, there exists output @out such that
(@in,@out) satisfies all rules in @rs

Consistent-thm : $\forall rs \cdot rs \subseteq 1 \mathinner{..} size \Rightarrow ($
$rs \in Consistent \Leftrightarrow (\forall in \cdot \exists out \cdot \forall i \cdot i \in rs \Rightarrow in \mapsto out \in rsat(i))$
$)$

An alternative (equivalent) expression for rules' consistency

`MIses_def` : $MISes = \{rs | rs \subseteq 1 \mathinner{..} size \wedge$
$rs \notin Consistent \wedge$
$(\forall s \cdot s \subset rs \Rightarrow s \in Consistent)$
$\}$

A set of rules @rs is minimal inconsistent iff:

1. @rs is inconsistent
2. any proper subset of @rs is consistent

END

An Event-B Specification of m0-enumerate_MIS
Creation Date: 4Dec2014 @ 10:09:11 AM

MACHINE m0-enumerate_MIS

This machine specifies the problem of enumerating MISes

SEES c0_3-MISes

VARIABLES

$mises$ The set of MISes found so far

INVARIANTS

mises-consistency : $mises \subseteq MISes$

The set of found MISes is the subset of all MISes

DLF : $(\exists mis \cdot mis \in MISes \land mis \notin mises) \lor mises = MISes$

Deadlock-freeness theorem

EVENTS

Initialisation

begin

act1 : $mises := \varnothing$

end

Event *find_MIS* $\widehat{=}$

A new MIS (@mis) is found

Status convergent

any

mis

where

grd1 : $mis \in MISes$

@mis is a MIS

grd2 : $mis \notin mises$

@mis has not yet been found

then

act1 : $mises := mises \cup \{mis\}$

Add @mis to the set of found MISes

end

Event *find_all_MIS* $\widehat{=}$

An observer events when all MISes are found

when

grd1 : $mises = MISes$

All MISes are found

then

skip

end

VARIANT

$MISes \setminus mises$ The variant for the convergence of event find_MIS is the set of MISes that haven't been found

END

An Event-B Specification of c1_0-iSAT
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c1_0-iSAT

This context abstractly defines the procedure for checking if a set of input constraints is SATisfiable

EXTENDS c0_1-iConstraints

CONSTANTS

$iSAT$ The input SAT procedure

AXIOMS

iSAT-def : $iSAT = \{is | is \subseteq 1 \mathinner{..} size \wedge (\exists in \cdot \forall i \cdot i \in is \Rightarrow in \in iConstraints(i))\}$

A set of input predicates @is is SATisfiable if there exists some input @in satisfies all constraints @i in @is

iSAT-antimonotonic : $\forall s, rs \cdot s \subseteq rs \wedge rs \in iSAT \Rightarrow s \in iSAT$

iSAT is anti-monotomic (with respect to the set-inclusion relationship)

END

An Event-B Specification of c1_0-oSAT
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c1_0-oSAT

This context abstractly defines the procedure for checking if a set of output constraints is SATisfiable

EXTENDS c0_1-oConstraints

CONSTANTS

$oSAT$ The output SAT procedure

AXIOMS

oSAT-def : $oSAT = \{os | os \subseteq 1 \mathinner{..} size \wedge (\exists out \cdot \forall i \cdot i \in os \Rightarrow out \in oConstraints(i))\}$

A set of output predicates @os is SATisfiable if there exists some output @out satisfies all constraints @i in @os

oSAT-antimonotonic : $\forall s, rs \cdot s \subseteq rs \wedge rs \in oSAT \Rightarrow s \in oSAT$

oSAT is anti-monotomic

(with respect to the set-inclusion relationship)

END

An Event-B Specification of c1_1-iSAT_and_oSAT_and_Consistent
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c1_1-iSAT_and_oSAT_and_Consistent

This context declares some relationships between input/output constraints' satisfiability and rules' consistency

EXTENDS c0_3-MISes

AXIOMS

Consistent_def : $Consistent = \{rs | rs \subseteq 1 \mathinner{..} size \wedge (\forall in \cdot \exists out \cdot \forall i \cdot i \in rs \Rightarrow in \mapsto out \in rsat(i))\}$

Copy of the axiom

rsat-def : $rsat = (\lambda i \cdot i \in 1 \mathinner{..} size \mid \{in \mapsto out | in \notin iConstraints(i) \vee out \in oConstraints(i)\})$

Copy of the axiom

oSAT_def : $oSAT = \{os | os \subseteq 1 \mathinner{..} size \wedge (\exists out \cdot \forall i \cdot i \in os \Rightarrow out \in oConstraints(i))\}$

Copy of the axiom

iSAT_def : $iSAT = \{is | is \subseteq 1 \mathinner{..} size \wedge (\exists in \cdot \forall i \cdot i \in is \Rightarrow in \in iConstraints(i))\}$

Copy of the axiom

iSAT_and_oUNSAT_and_Inconsistent : $\forall rs \cdot rs \in iSAT \wedge rs \notin oSAT \Rightarrow rs \notin Consistent$

Lemma 1:
A set of rule @rs is inconsistent if
(1) its input constraints are SATisfiable
(2) its output constraints are UNSATisfiable

oSAT_and_Consistent : $\forall rs \cdot rs \in oSAT \Rightarrow rs \in Consistent$

Lemma 2:

A set of rule @rs is consistent if
its output constraints are SATisfiable

END

An Event-B Specification of c1_1-oMUSes
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c1_1-oMUSes

This context defines the set of output MUSes

EXTENDS c1_0-oSAT

CONSTANTS

$oMUSes$

AXIOMS

oMUSes-def : $oMUSes = \{os | os \subseteq 1 \mathinner{..} size \land os \notin oSAT \land (\forall s \cdot s \subset os \Rightarrow s \in oSAT)\}$

A set of output constraints @os is a MUS iff:
1. @os is UNSATifiable
2. Any proper subset @s of @os is SATifiable

END

An Event-B Specification of c1_2-oMUSes_and_iSAT_and_MISes
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c1_2-oMUSes_and_iSAT_and_MISes

This context declares the main relationship between rules' MISes, output constraints' MUSes and input constraints' satisfiability

EXTENDS c1_1-iSAT_and_oSAT_and_Consistent

AXIOMS

iSAT_and_oUNSAT_and_Inconsistent : $\forall rs \cdot rs \in iSAT \land rs \notin oSAT \Rightarrow rs \notin Consistent$

Copy of the theorem

oSAT_and_Consistent : $\forall rs \cdot rs \in oSAT \Rightarrow rs \in Consistent$

Copy of the theorem

oMUSes_and_iSAT_and_MISes : $\forall rs \cdot rs \in MISes \Leftrightarrow rs \in oMUSes \land rs \in iSAT$

Theorem 1:
A set of rule @rs is a MIS iff
(1) its output constraints are a MUS
(2) its input constraints are satisfiable

oMUSes_and_iSAT_and_MISes_thm : $MISes = \{rs | rs \in oMUSes \land rs \in iSAT\}$

END

An Event-B Specification of m1-oMUSes_and_iSAT
Creation Date: 4Dec2014 @ 10:09:11 AM

MACHINE m1-oMUSes_and_iSAT

Introduce the algorithm using Minimal Unsatisfiable Subsets (MUSes) and SATifiable

REFINES m0-enumerate_MIS

SEES c1_2-oMUSes_and_iSAT_and_MISes

VARIABLES

$omuses$

mus

$findMUS\_pc$

INVARIANTS

omuses-consistency : $omuses \subseteq oMUSes$

findMUS_pc-type : $findMUS\_pc = 0 \lor findMUS\_pc = 1$

omuses_mises-consistency_0 : $findMUS\_pc = 0 \Rightarrow mises = omuses \cap MISes$

omuses_mises-consistency_1 : $findMUS\_pc = 1 \Rightarrow mises = (omuses \setminus \{mus\}) \cap MISes$

mus_omuses-consistency_1 : $findMUS\_pc = 1 \Rightarrow mus \in omuses$

DLF : $(\exists s \cdot findMUS\_pc = 0 \land s \in oMUSes \land s \notin omuses) \lor$
$(\exists mis \cdot findMUS\_pc = 1 \land mus \in iSAT \land mis = mus) \lor$
$(findMUS\_pc = 1 \land mus \notin iSAT) \lor$
$(findMUS\_pc = 0 \land omuses = oMUSes)$

EVENTS

Initialisation

begin

act1 : $omuses := \varnothing$

act2 : $mus :\in \mathbb{P}(\mathbb{Z})$

act4 : $findMUS\_pc := 0$

end

Event *find_MUS* $\widehat{=}$

Status convergent

any

s

where

grd1 : $findMUS\_pc = 0$

grd2 : $s \in oMUSes$

grd3 : $s \notin omuses$

then

act1 : $mus := s$

act2 : $omuses := omuses \cup \{s\}$

act3 : $findMUS\_pc := 1$

end

Event *find_MIS* $\widehat{=}$

Status convergent

refines *find_MIS*

any

mis

where

grd1 : $findMUS\_pc = 1$
grd2 : $mus \in iSAT$
grd3 : $mis = mus$
oMUSes_and_iSAT_and_MISes_thm : $MISes = \{rs | rs \in oMUSes \wedge rs \in iSAT\}$
omuses_mises-consistency : $mises = (omuses \setminus \{mus\}) \cap MISes$
thm1 : $(omuses \setminus \{mus\}) \cup \{mus\} = omuses$
thm2 : $MISes \cup \{mus\} = MISes$

then

act1 : $findMUS\_pc := 0$

end

Event *find_MUS_but_not_MIS* $\widehat{=}$
Status anticipated

when

grd1 : $findMUS\_pc = 1$
grd2 : $mus \notin iSAT$
oMUSes_and_iSAT_and_MISes : $MISes = \{rs | rs \in oMUSes \wedge rs \in iSAT\}$

then

act1 : $findMUS\_pc := 0$

end

Event *find_all_MIS* $\widehat{=}$
refines *find_all_MIS*

when

grd1 : $findMUS\_pc = 0$
grd2 : $omuses = oMUSes$
oMUSes_and_iSAT_and_MISes : $MISes = \{rs | rs \in oMUSes \wedge rs \in iSAT\}$
omuses_mises-consistency : $mises = omuses \cap MISes$

then

$skip$

end

VARIANT

$oMUSes \setminus omuses$

END

An Event-B Specification of m1-oMUSes
Creation Date: 4Dec2014 @ 10:09:11 AM

MACHINE m1-oMUSes

This machine is the result of decompose m1-oMUSes_and_iSAT

SEES c1_2-oMUSes_and_iSAT_and_MISes

VARIABLES

$omuses$

mus

INVARIANTS

omuses-type : $omuses \subseteq oMUSes$

mus-type : $mus \in \mathbb{P}(\mathbb{Z})$

EVENTS

Initialisation

begin

act1 : $omuses := \varnothing$

act2 : $mus :\in \mathbb{P}(\mathbb{Z})$

end

Event *find_MUS* $\widehat{=}$

any

s

where

grd1 : $s \in oMUSes$

grd2 : $s \notin omuses$

then

act1 : $mus := s$

act2 : $omuses := omuses \cup \{s\}$

end

Event *find_MIS* $\widehat{=}$

any

mis

where

grd1 : $mus \in iSAT$

grd2 : $mis = mus$

then

$skip$

end

Event *find_MUS_but_not_MIS* $\widehat{=}$

when

grd1 : $mus \notin iSAT$

then

$skip$

end

Event *find_all_MIS* $\widehat{=}$

when

grd1 : $omuses = oMUSes$

then

$skip$

end

END

An Event-B Specification of c2_0-oMSSes
Creation Date: 4Dec2014 @ 10:09:11 AM

CONTEXT c2_0-oMSSes

This context defines the set of output MSSes

EXTENDS c1_0-oSAT

CONSTANTS

$oMSSes$

AXIOMS

oMSSes-def : $oMSSes = \{os | os \subseteq 1 \mathinner{..} size \land os \in oSAT \land (\forall s \cdot s \subseteq 1 \mathinner{..} size \land os \subset s \Rightarrow s \notin oSAT)\}$

A set of output constraints @os is an MSS iff:

1. @os is SATifiable
2. Any proper superset @s of @os is UNSATifiable

END

An Event-B Specification of m2-MARCO
Creation Date: 4Dec2014 @ 10:09:11 AM

MACHINE m2-MARCO

This machine specifies the MARCO algorithm for enumerating MUSes

REFINES m1-oMUSes

SEES c1_2-oMUSes_and_iSAT_and_MISes, c2_0-oMSSes

VARIABLES

$omuses$

mus

$PSMan$

$seed$

$omsses$

pc

INVARIANTS

omsses-type : $omsses \subseteq oMSSes$

omuses_and_omsses_and_PSMan : $PSMan = \{rs | rs \subseteq 1 \mathinner{..} size \wedge (\exists s \cdot s \in omuses \wedge s \subseteq rs)\} \cup \{rs | rs \subseteq 1 \mathinner{..} size \wedge (\exists s \cdot s \in omsses \wedge rs \subseteq s)\}$

omuses_and_PSMan_and_oMUSes : $omuses = PSMan \cap oMUSes$

pc-type : $pc \in 0 \mathinner{..} 1$

seed-consistency : $pc = 1 \Rightarrow seed \subseteq 1 \mathinner{..} size \wedge seed \notin PSMan$

EVENTS

Initialisation

extended

begin

act1 : $omuses := \varnothing$

act2 : $mus :\in \mathbb{P}(\mathbb{Z})$

act3 : $PSMan := \varnothing$

act4 : $seed :\in \mathbb{P}(1 \mathinner{..} size)$

act5 : $omsses := \varnothing$

act6 : $pc := 0$

end

Event *getSet* $\widehat{=}$

any s

where

grd1 : $pc = 0$

grd2 : $s \in \mathbb{P}(1 \mathinner{..} size)$

grd3 : $s \notin PSMan$

then

act1 : $pc := 1$

act2 : $seed := s$

end

Event *find_MUS* $\widehat{=}$
refines *find_MUS*
any
s
where
grd1 : $pc = 1$
grd2 : $seed \notin oSAT$
grd3 : $s \in oMUSes$
Shrink to find MUS
grd4 : $s \subseteq seed$
omuses_and_omsses_and_PSMan : $PSMan = \{rs | rs \subseteq 1..size \wedge (\exists s \cdot s \in omuses \wedge s \subseteq rs)\} \cup \{rs | rs \subseteq 1 \mathinner{..} size \wedge (\exists s \cdot s \in omsses \wedge rs \subseteq s)\}$
then
act1 : $pc := 0$
act2 : $mus := s$
act3 : $omuses := omuses \cup \{s\}$
act4 : $PSMan := PSMan \cup \{rs | rs \subseteq 1 \mathinner{..} size \wedge s \subseteq rs\}$
end
Event *find_MSS* $\widehat{=}$
any
mss
where
grd1 : $pc = 1$
grd4 : $seed \in oSAT$
grd2 : $mss \in oMSSes$
Grow to find MSS
grd3 : $seed \subseteq mss$
omuses_and_omsses_and_PSMan : $PSMan = \{rs | rs \subseteq 1..size \wedge (\exists s \cdot s \in omuses \wedge s \subseteq rs)\} \cup \{rs | rs \subseteq 1 \mathinner{..} size \wedge (\exists s \cdot s \in omsses \wedge rs \subseteq s)\}$
oMSSes-def : $oMSSes = \{os | os \subseteq 1 \mathinner{..} size \wedge os \in oSAT \wedge (\forall s \cdot s \subseteq 1 \mathinner{..} size \wedge os \subset s \Rightarrow s \notin oSAT)\}$
then
act1 : $pc := 0$
act2 : $omsses := omsses \cup \{mss\}$
act3 : $PSMan := PSMan \cup \{rs | rs \subseteq 1 \mathinner{..} size \wedge rs \subseteq mss\}$
end
Event *find_MIS* $\widehat{=}$
extends *find_MIS*
any
mis

where

grd1 : $mus \in iSAT$
grd2 : $mis = mus$

then

$skip$

end

Event *find_MUS_but_not_MIS* $\widehat{=}$
extends *find_MUS_but_not_MIS*

when

grd1 : $mus \notin iSAT$

then

$skip$

end

Event *find_all_MIS* $\widehat{=}$
refines *find_all_MIS*

when

grd1 : $PSMan = \mathbb{P}(1 \mathinner{..} size)$
oMUSes-def : $oMUSes = \{os | os \subseteq 1 \mathinner{..} size \land$
$os \notin oSAT \land$
$(\forall s \cdot s \subset os \Rightarrow s \in oSAT)$
$\}$
omuses_and_PSMan_and_oMUSes : $omuses = PSMan \cap oMUSes$

then

$skip$

end

END

An Event-B Specification of m2-MUSesHunter
Creation Date: 4Dec2014 @ 10:09:11 AM

MACHINE m2-MUSesHunter

This machine specifies the MUSesHunter algorithm for enumerating MUSes

REFINES m1-oMUSes

SEES c1_2-oMUSes_and_iSAT_and_MISes, c2_0-oMSSes

VARIABLES

$omuses$

mus

$PSMan$

$seed$

$omsses$

pc

INVARIANTS

omsses-type : $omsses \subseteq oMSSes$

omuses_and_omsses_and_PSMan : $PSMan = \{rs | rs \subseteq 1 \mathinner{..} size \land (\exists s \cdot s \in omuses \land s \subseteq rs)\} \cup \{rs | rs \subseteq 1 \mathinner{..} size \land (\exists s \cdot s \in omuses \land rs \subset s)\} \cup \{rs | rs \subseteq 1 \mathinner{..} size \land (\exists s \cdot s \in omsses \land rs \subseteq s)\}$

omuses_and_PSMan_and_oMUSes : $omuses = PSMan \cap oMUSes$

pc-type : $pc \in 0 \mathinner{..} 1$

seed-consistency : $pc = 1 \Rightarrow seed \subseteq 1 \mathinner{..} size \land seed \notin PSMan$

SAT-shrink : $\forall S, T \cdot S \subseteq 1 \mathinner{..} size \land T \subseteq 1 \mathinner{..} size \land S \notin PSMan \land T \subseteq S \land T \in PSMan \Rightarrow T \in oSAT$

Lemma 3

UNSAT-grow : $\forall S, T \cdot S \subseteq 1 \mathinner{..} size \land T \subseteq 1 \mathinner{..} size \land S \notin PSMan \land S \subseteq T \land T \in PSMan \Rightarrow T \notin oSAT$

Lemma 4

EVENTS

Initialisation

extended

begin

act1 : $omuses := \varnothing$

act2 : $mus :\in \mathbb{P}(\mathbb{Z})$

act3 : $PSMan := \varnothing$

act4 : $seed :\in \mathbb{P}(1 \mathinner{..} size)$

act5 : $omsses := \varnothing$

act6 : $pc := 0$

end

Event *getSet* $\widehat{=}$

any

s

where

grd1 : $pc = 0$
grd2 : $s \in \mathbb{P}(1 \mathinner{..} size)$
grd3 : $s \notin PSMan$

then

act1 : $pc := 1$
act2 : $seed := s$

end

Event *find_MUS* $\widehat{=}$
refines *find_MUS*

any

s

where

grd1 : $pc = 1$
grd2 : $seed \notin oSAT$
grd3 : $s \in oMUSes$

Shrink to find MUS

grd4 : $s \subseteq seed$
omuses_and_omsses_and_PSMan : $PSMan = \{rs | rs \subseteq 1..size \wedge (\exists s \cdot s \in omuses \wedge s \subseteq rs)\} \cup \{rs | rs \subseteq 1 \mathinner{..} size \wedge (\exists s \cdot s \in omuses \wedge rs \subset s)\} \cup \{rs | rs \subseteq 1 \mathinner{..} size \wedge (\exists s \cdot s \in omsses \wedge rs \subseteq s)\}$

then

act1 : $pc := 0$
act2 : $mus := s$
act3 : $omuses := omuses \cup \{s\}$
act4 : $PSMan := PSMan \cup \{rs | rs \subseteq 1..size \wedge s \subseteq rs\} \cup \{rs | rs \subseteq 1..size \wedge rs \subset s\}$

end

Event *find_MSS* $\widehat{=}$

any

mss

where

grd1 : $pc = 1$
grd2 : $seed \in oSAT$
grd4 : $mss \in oMSSes$

Grow to find MSS

grd5 : $seed \subseteq mss$
omuses_and_omsses_and_PSMan : $PSMan = \{rs | rs \subseteq 1..size \wedge (\exists s \cdot s \in omuses \wedge s \subseteq rs)\} \cup \{rs | rs \subseteq 1 \mathinner{..} size \wedge (\exists s \cdot s \in omuses \wedge rs \subset s)\} \cup \{rs | rs \subseteq 1 \mathinner{..} size \wedge (\exists s \cdot s \in omsses \wedge rs \subseteq s)\}$

$oMSSes\text{-}def$: $oMSSes = \{os | os \subseteq 1 \mathinner{..} size \land$
$os \in oSAT \land$
$(\forall s \cdot s \subseteq 1 \mathinner{..} size \land os \subset s \Rightarrow s \notin oSAT)$
$\}$

then

act1 : $pc := 0$
act2 : $omsses := omsses \cup \{mss\}$
act3 : $PSMan := PSMan \cup \{rs | rs \subseteq 1 \mathinner{..} size \land rs \subseteq mss\}$

end

Event *find_new_seed* $\widehat{=}$

any

new_seed

where

grd1 : $pc = 1$
grd2 : $seed \in oSAT$
grd3 : $seed \subset new\_seed$
grd4 : $new\_seed \subseteq 1 \mathinner{..} size$
grd5 : $new\_seed \notin oSAT$
grd6 : $new\_seed \notin PSMan$

then

act1 : $seed := new\_seed$

end

Event *find_MIS* $\widehat{=}$
extends *find_MIS*

any

mis

where

grd1 : $mus \in iSAT$
grd2 : $mis = mus$

then

$skip$

end

Event *find_MUS_but_not_MIS* $\widehat{=}$
extends *find_MUS_but_not_MIS*

when

grd1 : $mus \notin iSAT$

then

$skip$

end

Event *find_all_MIS* $\widehat{=}$
refines *find_all_MIS*

when

grd1 : $PSMan = \mathbb{P}(1 \mathinner{..} size)$

$oMUSes\text{-}def$: $oMUSes = \{os | os \subseteq 1 \,..\, size \land$
$os \notin oSAT \land$
$(\forall s \cdot s \subset os \Rightarrow s \in oSAT)$
$\}$
$omuses\_and\_PSMan\_and\_oMUSes$: $omuses = PSMan \cap oMUSes$

then

$skip$

end

END