Minimal Strongly Unsatisfiable Subsets of Reactive System Specifications

Shigeki Hagihara
Dept. of Computer Science
Tokyo Inst. of Technology
hagihara@fmx.cs.titech.ac.jp

Naoki Egawa
R&D Division, Sony
Computer Entertainment Inc.
negawa@rd.scei.sony.co.jp

Masaya Shimakawa
Dept. of Computer Science
Tokyo Inst. of Technology
masaya@fmx.cs.titech.ac.jp

Naoki Yonezaki
Dept. of Computer Science
Tokyo Inst. of Technology
yonezaki@cs.titech.ac.jp

ABSTRACT

Verifying realizability in the specification phase is expected to reduce the development costs of safety-critical reactive systems. If a specification is not realizable, we must correct the specification. However, it is not always obvious what part of a specification should be modified. In this paper, we propose a method for obtaining the location of flaws. Rather than realizability, we use strong satisfiability, due to the fact that many practical unrealizable specifications are also strongly unsatisfiable. Using strong satisfiability, the process of analyzing realizability becomes less complex. We define minimal strongly unsatisfiable subsets (MSUSs) to locate flaws, and construct a procedure to compute them. We also show correctness properties of our method, and clarify the time complexity of our method. Furthermore, we implement the procedure, and confirm that MSUSs are computable for specifications of reactive systems at non-trivial scales.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques*

General Terms

Verification; Reliability; Algorithms

Keywords

Reactive systems; Specifications; Realizability; Strong satisfiability; Flaw-location analysis; LTL; Büchi automata; Minimal unsatisfiable subsets

1. INTRODUCTION

Reactive systems are widespread, and include systems that control elevators, vending machines, nuclear power plants, and air traffic. Reactive systems respond to requests from the environment at an appropriate time. When designing a system of this kind, requirements are analyzed and then described as a specification for the system. If the specification has a flaw, such as inappropriate case splitting, the system may display unintended behavior. Indeed, many fatal accidents in safety critical reactive systems have occurred due to the rise of such unexpected situations. Therefore, it is important to ensure that specifications do not possess such flaws.

More precisely, a reactive system specification must have a model that can respond to any possible set of requests at an appropriate time. This property is called realizability, and was proposed in [1, 11]. Flaws in specifications can be classified as follows.

Case 1: Specifications are realizable but involve flaws.

- There are critical situations that were not considered during the design and testing phases, and thus the specifications do not define behavior for those situations.
- The intent of a given specification is incorrect and does not lead to the expected outcome.

Case 2: The specifications are not realizable.

- A specification cannot achieve realizability because its intent is not expressed correctly.
- A specification cannot achieve realizability due to unintended conflicts between parts of the specification.

In this paper, we consider the type of flaws in Case 2. If a specification is found to possess such flaws, they must be located and eliminated. If it is possible to locate the flaws, it should be straightforward for system designers to correct them.

Here we propose a method for locating flaws in a specification. Rather than realizability, we use strong satisfiability, which is a necessary precondition for realizability. Many practical unrealizable specifications are also strongly unsatisfiable, and strong satisfiability may lead to less complex analysis than realizability. We define minimal strongly unsatisfiable subsets as an indication of the location of flaws, and detail a procedure to compute them. A minimal strongly unsatisfiable subset represents a minimal part that is not strongly satisfiable in the specification. In this procedure, we take a specification and an input event sequence from an environment as inputs, and then compute all of the minimal strongly unsatisfiable subsets by analyzing a graph representing behavior for the input event sequences on the ω-automaton representing the specification. We show correctness properties of our method: soundness, and completeness, and clarify the time complexity of our method. Furthermore,

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642968 .

we report an implementation of the procedure, and confirm that minimal strongly unsatisfiable subsets are computable for the specifications of reactive systems at non-trivial scales.

2. SPECIFICATIONS OF REACTIVE SYSTEMS AND THEIR PROPERTIES

2.1 Reactive systems

A reactive system responds to requests from an environment with appropriate timing.

DEFINITION 1 (REACTIVE SYSTEM). *A reactive system RS is a tuple $\langle X, Y, r\rangle$, where X is a set of events caused by an environment, Y is a set of events caused by the system, and $r : (2^X)^+ \mapsto 2^Y$ is a reaction function.*

We call events caused by the environment 'environmental events,' and events caused by the system 'system events.' The set $(2^X)^+$ is the set of all finite sequences of sets of environmental events. A reaction function r relates sequences of sets of past environmental events with the set of current system events.

2.2 Language for reactive system specifications

The timing of environmental and system events is an essential element of reactive systems. In this paper, we use linear temporal logic (LTL) to describe the specifications of reactive systems. In LTL, in addition to the operators $\wedge$, $\vee$, $\rightarrow$, $\neg$, $\top$ and $\bot$, we can use the temporal operators $\bigcirc$ and U. The notation $\bigcirc f$ states 'f holds at the next time,' whereas fUg represents 'f always holds until g holds.' The notations fRg, $\Diamond f$, and $\Box f$ are abbreviations for $\neg(\neg fU\neg g)$, $\top Uf$, and $\neg\Diamond\neg f$ respectively. We treat environmental events and system events as atomic propositions.

Behavior σ is an infinite sequence of sets of events. $\sigma \models f$ represents that σ satisfies f, which is defined as usual.

2.3 Properties of reactive system specifications

When developing reactive system specifications, it is important to satisfy realizability. Realizability requires that there exists a reactive system such that for any environmental events of any timing, the reactive system produces system events such that the specification holds.

DEFINITION 2 (REALIZABILITY). *A specification φ is realizable if $\exists RS\forall\tilde{a}(behave_{RS}(\tilde{a}) \models \varphi)$ holds, where $\tilde{a}$ is an infinite sequence of sets of environmental events, i.e., $\tilde{a} \in (2^X)^\omega$. $behave_{RS}(\tilde{a})$ is the infinite behavior by RS for $\tilde{a}$, and is defined as follows. If $\tilde{a} = a_0a_1\ldots$, $behave_{RS}(\tilde{a}) = (a_0\cup b_0)(a_1\cup b_1)\ldots$, where b_i is a set of system events caused by RS, i.e., $b_i = r(a_0\ldots a_i)$, and $\cup$ is the union operator of two sets.*

The following property is a necessary condition for realizability [10].

DEFINITION 3 (STRONG SATISFIABILITY). *A specification φ is strongly satisfiable if $\forall\tilde{a}\exists\tilde{b}(\langle\tilde{a},\tilde{b}\rangle \models \varphi)$ holds, where $\tilde{b}$ is an infinite sequence of sets of system events, i.e., $\tilde{b} \in (2^Y)^\omega$. If $\tilde{a} = a_0a_1\ldots$ and $\tilde{b} = b_0b_1\ldots$, then $\langle\tilde{a},\tilde{b}\rangle$ is defined as $\langle\tilde{a},\tilde{b}\rangle = (a_0\cup b_0)(a_1\cup b_1)\ldots$.*

Intuitively, strong satisfiability is the property that, if an infinite sequence of future environmental events is known, then the system can determine an infinite sequence of sets of system events.

Strong satisfiability is a necessary condition for realizability, i.e., all realizable specifications are strongly satisfiable. However, many practical strongly satisfiable specifications are also realizable. Strong satisfiability requires a less complex analysis compared to realizability. Checking for strong satisfiability is EXPSPACE-complete [13], whereas checking realizability is 2EXPTIME-complete [12].

If a specification is not strongly satisfiable, then there exists a counterexample that is evidence for unsatisfiability of the specification.

DEFINITION 4 (COUNTEREXAMPLE). *Let φ be a specification that is not strongly satisfiable, that is, φ satisfies $\exists\tilde{a}\forall\tilde{b}(\langle\tilde{a},\tilde{b}\rangle \not\models \varphi)$. Then, a counterexample is $\tilde{a}$, which satisfies $\forall\tilde{b}(\langle\tilde{a},\tilde{b}\rangle \not\models \varphi)$.*

EXAMPLE 1. *Let us consider the following reactive system specification $\varphi : \bigwedge_{1\leq i\leq 3}\varphi_i$, $\varphi_1 : \Box(x_1 \rightarrow \Diamond y)$, $\varphi_2 : \Box(x_2 \rightarrow \neg y)$, $\varphi_3 : \Box((x_3 \wedge y) \rightarrow (yUx_2))$, where x_1, x_2, and x_3 are environmental events, and y is a system event. Each of $\varphi_1, \varphi_2, \varphi_3$ is strongly satisfiable individually; however, φ is not strongly satisfiable, because there is no response that satisfies φ for the environmental behavior where x_2 continues to occur after x_1 occurs. Formally, this is because for $\tilde{a} = \{x_2\}\{x_2, x_3\}(\{x_1, x_2\}\{x_2, x_3\})^\omega$, $\exists\tilde{b}(\langle\tilde{a},\tilde{b}\rangle \models \varphi)$ does not hold. $\tilde{a}$ is a counterexample that shows unsatisfiability of φ.*

3. MINIMAL STRONGLY UNSATISFIABLE SUBSETS

In this section, we define minimal strongly unsatisfiable subsets (MSUSs), and propose a procedure to compute them.

3.1 Minimal Strongly Unsatisfiable Subsets

Intuitively, an MSUS represents a minimal part of a specification that is not strongly satisfiable.

DEFINITION 5 (STRONGLY UNSATISFIABLE SUBSET, MSUS). *Let $\varphi : \bigwedge_{i\in I}\varphi_i$ be a strongly unsatisfiable specification, and let $\tilde{a}$ be a counterexample, i.e., an infinite sequence of sets of environmental events that satisfy $\forall\tilde{b}(\langle\tilde{a},\tilde{b}\rangle \not\models \varphi)$. If a set of indices $I' \subseteq I$ satisfies $\forall\tilde{b}(\langle\tilde{a},\tilde{b}\rangle \not\models \bigwedge_{i\in I'}\varphi_i)$, we say I' is a strongly unsatisfiable subset in φ for $\tilde{a}$. If I' is minimal, we say I' is a minimal strongly unsatisfiable subset (MSUS) in φ for $\tilde{a}$.*

EXAMPLE 2. *Let us consider φ in Example 1. The strongly unsatisfiable subsets for the counterexample $\tilde{a} = \{x_2\}\{x_2, x_3\}(\{x_1, x_2\}\{x_2, x_3\})^\omega$ are $\{1, 2\}$ and $\{1, 2, 3\}$. MSUS is $\{1, 2\}$. Any other set of indices J satisfies $\exists\tilde{b}(\langle\tilde{a},\tilde{b}\rangle \models \bigwedge_{i\in J}\varphi_i)$. This is because $\bigwedge_{i\in J}\varphi_i$ holds by considering $\tilde{b}$ where y keeps on occurring for $J = \{1, 3\}$, and by considering $\tilde{b}$ where $\neg y$ keeps on occurring for $J = \{2, 3\}$. This means that the collision of subspecifications φ_1 and φ_2 results in φ being strongly unsatisfiable.*

3.2 Preliminary to procedure definition

Büchi Automata. Our method of computing all of the MSUSs uses ω-automata, which accepts only the same set of behavior as that which satisfies a specification. In this paper, we use Büchi automata and generalized Büchi automata, of which the edges are labeled by Boolean formulae as follows.

DEFINITION 6 (BÜCHI AUTOMATA). *Let P be a set of propositions, i.e., environmental or system events. A Büchi automaton on an alphabet 2^P is defined by $A = \langle Q, q_0, \delta, F\rangle$, where Q is a finite set of states, q_0 is an initial state, $\delta \subseteq Q \times B(P) \times Q$ is a transition relation, $F \subseteq Q$ is a set of final states, and $B(P)$ is a set of Boolean formulae that consist of propositions in P and connectives $\neg, \vee, \wedge$. A run of A on an ω-word $\alpha = \alpha[0]\alpha[1]\ldots$ is an infinite sequence $\gamma = \gamma[0]\gamma[1]\ldots$ of states, where $\gamma[0] = q_0$ and $(\gamma[i], b_i, \gamma[i+1]) \in \delta$ and $\alpha[i] \models b_i$ for some b_i for all $i \geq 0$. If $In(\gamma) \cap F \neq \emptyset$ holds, a run γ is said to be successful, where $In(\gamma)$ is a set of states that occur infinitely often in γ. If there is a successful run of A on α, we say that A accepts α. A set of ω-words that are accepted by A is called the language accepted by A, which is represented by $L(A)$.*

DEFINITION 7 (GENERALIZED BÜCHI AUTOMATA). *A generalized Büchi automaton on an alphabet 2^P is defined by $A = \langle Q, \Sigma, \delta, I, \mathcal{F}\rangle$, where Q, Σ, δ, and I are defined as above for a Büchi automaton. $\mathcal{F} = \{F_1, \ldots, F_n\}$ is a set of sets of final states, and satisfies $F_i \subseteq Q$. If $\forall i($ $Inf(\gamma) \cap F_i \neq \emptyset)$ holds, a run γ is said to be successful. If there is a successful run of A on α, we say that A accepts α.*

Translation to Büchi automata from LTL formulae. There are numerous ways to translate an LTL formula into a Büchi automaton that accepts exactly the same set of behaviors satisfying the formula. Formally, translation methods convert an LTL formula φ to a Büchi automaton A which satisfy $L(A) = \{\sigma \mid \sigma \models \varphi\}$. Improvements to this procedure have been reported [3], and graphical tools to construct and manipulate automata have also been reported [14]. We obtain a Büchi automaton for a specification using the methods introduced above.

Product of automata. In our procedure, we compute a product of Büchi automata, defined as follows.

DEFINITION 8 (PRODUCT OF AUTOMATA $\otimes$). *Let $A_1, ..., A_n$ with $A_i = \langle Q_i, q_{0i}, \delta_i, F_i\rangle$ be Büchi automata. The product of $A_1, ..., A_n$ is the generalized Büchi automaton $\otimes_{1\leq i\leq n} A_i = \langle Q, q_0, \delta, \mathcal{F}\rangle$ defined as follows. $Q = Q_1 \times \ldots \times Q_n$, $q_0 = (q_{01}, \ldots, q_{0n})$, $\delta = \{((q_1, \ldots, q_n), b_1 \wedge \ldots \wedge b_n, (q'_1, \ldots, q'_n)) \mid \forall 1 \leq i \leq n(q_i, b_i, q'_i) \in \delta_i\}$, $\mathcal{F} = \{F'_1, \ldots, F'_n\}$, where $F'_i = \{(q_1, \ldots, q_n) \mid q_i \in F_i \wedge \forall j \neq i(q_j \in Q_j)\}$.*

REMARK 1. *Let A_{φ_i} be a Büchi automaton that satisfies $L(A_{\varphi_i}) = \{\sigma \mid \sigma \models \varphi_i\}$. Then, $L(\otimes_{1\leq i\leq n} A_{\varphi_i}) = \{\sigma \mid \sigma \models \varphi_1 \wedge \ldots \wedge \varphi_n\}$ holds.*

3.3 Procedure for computing MSUSs

In this section, we propose a method for computing all of the MSUSs in $\varphi : \bigwedge_{1\leq i\leq n} \varphi_i$.

The procedure takes as inputs (1) Büchi automata $A_1, \ldots, A_n$ that satisfy $L(A_i) = \{\sigma \mid \sigma \models \varphi_i\}$, and (2) a counterexample $\tilde{a}$, i.e., an infinite sequence of sets of environmental events that satisfies $\forall \tilde{b}(\langle \tilde{a}, \tilde{b}\rangle \not\models \varphi)$, and outputs the set of all of the MSUSs for $\tilde{a}$ in φ. We assume that each A_i is total, which means that for any state in A_i, any set of events, there is a transition from that state based on that set of events. Any Büchi automata can be converted into total automata without changing the accepted languages. We also assume that $\tilde{a}$ is given by $\bar{a_1}\bar{a_2}^\omega$, which is a concatenation of a finite sequence $\bar{a_1}$ of environmental events and an infinite iteration of a finite sequence $\bar{a_2}$ of environmental events.

Input: $\overline{\overline{A}} = (Q, q_0, \overline{\overline{\delta}}, \mathcal{F}), \tilde{a} = \bar{a_1}\bar{a_2}^\omega$
Output: Run Graph $G = (V, E)$
  $V := \{(q_0, \tilde{a})\}$, $V' := \emptyset$, $E := \emptyset$, $E' := \emptyset$
  while $V \neq V' \vee E \neq E'$ **do**
    $V' := V$, $E' := E$
    for each $(q, \tilde{z}) \in V$ **do**
      for each $(q, b, q') \in \overline{\overline{\delta}}$ **such that** $head(\tilde{z}) \models b$ **do**
        $V := V \cup \{(q', tail(\tilde{z}))\}$, $E := E \cup \{((q, \tilde{z}), (q', tail(\tilde{z})))\}$
      end for
    end for
  end while
  return (V, E)

(Here *head* and *tail* are functions that return the first element and the remaining elements, respectively.)

Figure 1: Algorithm for constructing a run graph.

STEP 1 We construct $\otimes_{1\leq i\leq n} A_i$ from $A_1, \ldots, A_n$. The resulting automaton is $A = \langle Q, q_0, \delta, \mathcal{F}\rangle$

STEP 2 We obtain a Büchi automaton $\overline{\overline{A}} = \langle Q, q_0, \overline{\overline{\delta}}, \mathcal{F}\rangle$ by restricting A to only environmental events, where $\overline{\overline{\delta}} = \{(q, E(b), q') \mid (q, b, q') \in \delta\}$, $E(b) = \bigvee_\theta b\theta$, θ is a substitution that substitutes $\top$ or $\bot$ for all of the system events in b. Intuitively, $E(b)$ denotes the quantified Boolean formula $\exists y_1 \ldots \exists y_n b$, if $y_1, \ldots, y_n$ are system events that appear in b. Note that all of the labels in $\overline{\overline{A}}$ are Boolean formulae that consist of only environmental events. This manipulation of automata is known as projection.

STEP 3 We construct a run graph $G = \langle V, E\rangle$ from $\overline{\overline{A}}$ and $\tilde{a} = \bar{a_1}\bar{a_2}^\omega$ using the algorithm shown in Fig. 1. G represents the set of runs of $\overline{\overline{A}}$ on $\tilde{a}$. Each node of G is a tuple, the first element is a state in $\overline{\overline{A}}$, and the second element is a remaining part of $\tilde{a}$ that has not yet been processed.

REMARK 2. $L(\overline{\overline{A}}) = \{\tilde{a} \mid \exists \tilde{b}(\langle \tilde{a}, \tilde{b}\rangle \models \varphi)\}$ *holds by the definition of $\overline{\overline{\delta}}$. A path of the first element of G coincides with a run of $\overline{\overline{A}}$ on $\tilde{a}$.*

STEP 4 We compute the maximal strongly connected components (SCCs) in G. Let SC be the resulting set of SCCs.

STEP 5 For each $sc \in SC$, we obtain I_{sc}, which is the set of indices of automata that do not include final states: $I_{sc} = \{k \mid \{q_k \mid ((q_1, \ldots, q_n), \tilde{a}') \in sc\} \cap F_k = \emptyset\}$, where $((q_1, \ldots, q_n), \tilde{a}')$ is a node included in sc. $(q_1, \ldots, q_n)$ is a state in $\overline{\overline{A}}$ and is represented by a tuple of states of $A_1, \ldots, A_n$.

STEP 6 We consider the set $\mathcal{I}$ of sets I of indices that satisfy $\forall sc \in SC(I \cap I_{sc} \neq \emptyset)$, and output the set $\mathcal{I}_{MSUS} \subseteq \mathcal{I}$, which consists of all of the minimal sets in $\mathcal{I}$.

3.4 Example application

We apply the procedure to the specification φ and the counterexample $\tilde{a}$ shown in Example 1. Büchi automata A_1, A_2, A_3 for $\varphi_1, \varphi_2, \varphi_3$ are obtained in advance as shown in Fig. 2(left). The states with the symbol $\triangleright$ are initial states, and those with the double circle are final states. The labels with edges are transition conditions, represented by Boolean formulae obtained by connecting literals horizontally with $\wedge$ and vertically with $\vee$. First, in STEP 1, we obtain A, which is the product of A_1, A_2, and A_3, and in STEP 2, we obtain $\overline{\overline{A}}$ by considering only environmental events, as shown in Fig. 2(center). Each number in tuples of states in $\overline{\overline{A}}$ corresponds to the indices of states in A_1, A_2, A_3. Next, in STEP

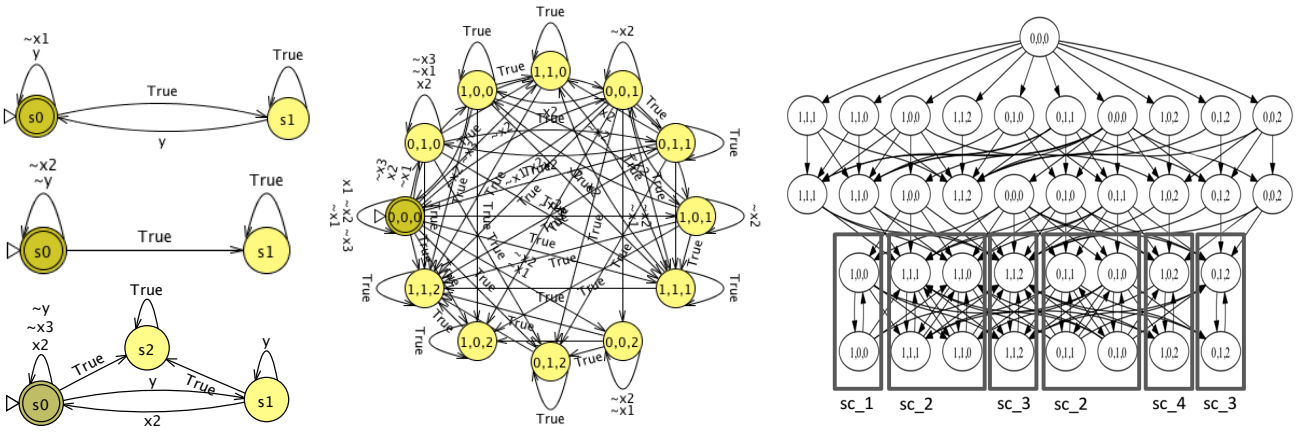

Figure 2: The automata of sub-specification A_1, A_2, A_3, the automaton $\overline{\overline{A}}$, and the run graph G.

3, we obtain the run graph G, shown in Fig. 2(right). Each path of G coincides with a run of $\overline{\overline{A}}$ on $\tilde{a}$. Although each node in G is a tuple, in Fig. 2(right), we show only the first element (i.e., a state in $\overline{\overline{A}}$) of the tuple, and arrange nodes with the same second elements in the same row. In STEP 4, we compute the SCCs and obtain $SC = \{sc_1, sc_2, sc_3, sc_4\}$ in G, shown in Fig. 2(right). In STEP 5, we obtain the set of indices of automata that do not include final states in the SCCs. Because each final state of A_1, A_2, A_3 is $s0$, if 0 is included in the tuple of states, the final state of the corresponding automaton is included in the SCC. For instance, the final states 0 of A_2 and A_3 are included in sc_1; however, the final states 0 of A_1 are not included in sc_1. We conclude that $I_{sc_1} = \{1\}$. Similarly, we conclude that $I_{sc_2} = \{2\}, I_{sc_3} = \{2,3\}, I_{sc_4} = \{1,3\}$. Finally, in STEP 6, we compute I, which satisfies $\forall sc \in SC(I \cap I_{sc} \neq \emptyset)$. In this case, $\{1,2\}$ and $\{1,2,3\}$ satisfy this condition, i.e., $\mathcal{I} = \{\{1,2\},\{1,2,3\}\}$. We obtain all of the minimal sets from $\mathcal{I}$. Hence, in this case, we obtain $\mathcal{I}_{MSUS} = \{\{1,2\}\}$, which is the same result as that given in Example 2.

4. CORRECTNESS AND COMPLEXITY

In this section, we show correctness properties of our procedure, i.e., soundness, and completeness. We also clarify the complexity of the procedure.

THEOREM 1 (SOUNDNESS, COMPLETENESS). *Let $\varphi : \bigwedge_{1\leq i\leq n}\varphi_i$ be a specification, $A_1, \ldots, A_n$ be Büchi automata that satisfy $L(A_i) = \{\sigma \mid \sigma \models \varphi_i\}$, $\tilde{a}$ be a counterexample, i.e., an infinite sequence of sets of environmental events that satisfy $\forall \tilde{b}(\langle \tilde{a}, \tilde{b}\rangle \not\models \varphi)$. If $\mathcal{I}_{MSUS}$ is a set of sets of indices that is the result of applying the procedure to $A_1, \ldots, A_n$ and $\tilde{a}$, then ($I \in \mathcal{I}_{MSUS} \Leftrightarrow I$ is a MSUS in φ for $\tilde{a}$) holds.*

This theorem can be proved by using Lemma 1. We omit the proof. Lemma 1 represents that the result of projecting $\overline{\overline{\otimes_{1\leq i\leq n}A_i}}$ by I can be considered as $\overline{\overline{\otimes_{i\in I}A_i}}$ which was not constructed in the procedure.

LEMMA 1. *$proj(Run(\overline{\overline{\otimes_{1\leq i\leq n}A_i}}, \sigma), I) = Run(\overline{\overline{\otimes_{i\in I}A_i}}, \sigma)$ holds, where $Run(A, \tilde{a})$ is the set of runs of A on $\tilde{a}$, $proj(Run(\overline{\overline{\otimes_{1\leq i\leq n}A_i}}, \tilde{a}), I)$ is the set of runs obtained by projecting $(q_1, \ldots, q_n)$ into $(q_{i_1}, \ldots, q_{i_m})$, where $I = \{i_1, \ldots, i_m\}$, and $(q_1, \ldots, q_n)$ is a tuple of states that appears in runs of $\overline{\overline{\otimes_{1\leq i\leq n}A_i}}$ on $\tilde{a}$.*

This lemma obviously holds because the following (I)(II)(III) can be proved. For proving (III), we use totality of A_i. We omit these proofs.

(I) $Run(\overline{\overline{A}}, \tilde{a}) = \bigcup_{\tilde{b}} Run(A, \langle \tilde{a}, \tilde{b}\rangle)$,
(II) $proj(Run(\overline{\overline{\otimes_{0<i\leq n}A_i}}, \tilde{a}), I)$
$= \bigcup_{\tilde{b}} proj(Run(\otimes_{0<i\leq n}A_i, \langle \tilde{a}, \tilde{b}\rangle), I)$,
(III) $proj(Run(\otimes_{0<i\leq n}A_i, \sigma), I) = Run(\otimes_{i\in I}A_i, \sigma)$.

We can clarify the time complexity of our procedure by estimating computation time for each step of our procedure.

THEOREM 2 (TIME COMPLEXITY). *Let $\varphi : \bigwedge_{1\leq i\leq n}\varphi_i$ be a specification, and let $\tilde{a} = \bar{a_1}(\bar{a_2})^\omega$ be a counterexample. We can translate $\varphi_1, \ldots, \varphi_n$ into a Büchi automaton $A_1, \ldots, A_n$, apply the procedure to $A_1, \ldots, A_n$ and $\tilde{a}$, and compute the set of MSUSs in time $2^{O(|\varphi|+\log(|\bar{a_1}|+|\bar{a_2}|)+\log n)} + 2^{O(2^n)}$.*

5. APPLICABILITY

Our method is useful for correcting strongly unsatisfiable reactive system specifications, as demonstrated by the following example of a control system for a door. The initial specification is as follows.

0. A door has three buttons: an open button, a close button, and an open-extension button.
1. If the open button is pushed, the door eventually opens.
2. While the close button is pressed, the door remains shut.
3. While the door is open, if the open-extension button is pushed, the door remains open until the close button is pushed.

The event 'the open (or close or open-extension) button is pushed,' is an environmental event. We use x_1 (x_2, x_3) to describe this event. The event 'the door is open (closed)' is a system event. We use y ($\neg y$) to describe this. This initial specification is represented by $\Box(x_1 \rightarrow \Diamond y) \wedge \Box(x_2 \rightarrow \neg y) \wedge \Box((x_3 \wedge y) \rightarrow (yUx_2))$ in LTL, which is identical to φ in Example 1. Because this specification is strongly unsatisfiable, as stated in Example 1, it is necessary to identify a flaw that causes strong unsatisfiability and correct this specification. To correct this specification, we use MSUSs in the specification. Strong satisfiability can be checked for using the method described in [6], and if a specification is

not strongly satisfiable, a counterexample of the form $\bar{a}\bar{a}'^{\omega}$ can be obtained. In this case, we suppose that the counterexample $\tilde{a}$ shown in Example 1 is obtained. The set of MSUSs $\{\{1, 2\}\}$ is obtained by applying the procedure to φ and $\tilde{a}$, as stated in Sect. 3.4. This states that, to make $\tilde{a}$ not be a counterexample, it is not necessary to modify φ_3 (3 in the specification), but it is necessary to modify φ_1 or φ_2 (1 or 2 in the specification). Several modifications are possible; for example, 2 in the initial specification may be weakened to 2', i.e.,

2'. If the close button is pushed, the door eventually closes.

The modified specification is represented by $\Box(x_1 \rightarrow \Diamond y) \wedge \Box(x_2 \rightarrow \Diamond\neg y) \wedge \Box((x_3 \wedge y) \rightarrow (yUx_2))$. For $\tilde{a}$, there is an infinite sequence of sets of system events that satisfies this modified specification. That is, following this modification, $\tilde{a}$ is no longer a counterexample. In this case, because only one MSUS $\{1, 2\}$ was obtained by the procedure, only one modification was required. In general, multiple MSUSs may be obtained. In such cases, multiple modifications may be required to account for all of the MSUSs.

Furthermore, one counterexample does not always correspond to all of the flaws that give rise to strong unsatisfiability of the specification. Although modification succeeded in making $\tilde{a}$ no longer a counterexample, flaws causing strong unsatisfiability of the specification may remain. In such cases, we can obtain strongly satisfiable specifications by iterating the following processes: (1) we determine whether strong satisfiability of the specification exists, and if not, we obtain a counterexample; (2) we compute the MSUSs; and (3) we correct the specification accordingly.

6. IMPLEMENTATION AND EVALUATION

6.1 Implementation

We implemented the procedure using Python 2.7. The inputs were a collection of Büchi automata representing specifications, and a counterexample. We used the tools GOAL[14] and LTL3BA[3] to translate the specification into Büchi automata.

6.2 Evaluation

We evaluated our implementation on a machine with a Core i7-3820 processor running at 3.60 GHz, with 32 GB of memory, and which was running the Ubuntu Linux operating system. We apply this implementation to a practical specification at a non-trivial scale. We consider the specification of an m-floor elevator system described in [2], shown in Fig. 3. The specification has $m+2$ environmental events, $2m+4$ system events, and $12m+8$ temporal formulae. The specification consists of (a) a specification for each floor, divided into functional requirements and non-functional requirements; (b) a specification for a door of the lift; and (c) a specification of the physical constraints. This specification is not strongly satisfiable, because if the button at a given floor is held (i.e., pushed continuously), the lift will stay at that floor and will not move to other floors. In this experiment, we use the counterexample representing the behavior that both buttons at the first and second floors are pushed continuously. We consider the following two kinds of partitions of specifications: A. (a1),...,(am),(b),(c), and B. (a1f),(a1n),...,(amf),(amn),(b),(c), where (ai) represents the specification for the i-th floor, (aif) and (ain) represent the functional and non-functional requirements, respectively, for the i-th floor specification. We show the results in Table 1. The execution time includes the time to translate the subspecifications in LTL into Büchi automata. If the execution does not terminate within 1800 seconds, we show the notation 't/o.' After partitioning the specification into 14 subspecifications (the bottommost case in Table 1), it failed to compute MSUSs within 1800 seconds, and the run graph was not constructed. Because we used a naive transformation algorithm and naive data structures to represent the run graph, the memory space for the run graph was large. Except for this case, in spite of the naive implementation, we obtained the intended results in a reasonable period of time. This indicates that the flaw did not appear in non-functional requirements, and appeared only in the functional requirements for the first and second floors, and the physical constraints. Hence, our method is useful not only for locating the flaws, but also for classifying subspecifications, including functional/non-functional requirements and physical requirements. We confirmed that the set of MSUSs can be computed from a practical specification at non-trivial scale in a reasonable amount of time, and found that the MSUS is useful for locating flaws in practical specifications.

m	partition	the set of MSUSs	time[s]
3	A	{{(a1),(a2),(c)}}	0.26
	B	{{(a1f),(a2f),(c)}}	1.18
4	A	{{(a1),(a2),(c)}}	1.77
	B	{{(a1f),(a2f),(c)}}	27.99
5	A	{{(a1),(a2),(c)}}	19.48
	B	{{(a1f),(a2f),(c)}}	853.65
6	A	{{(a1),(a2),(c)}}	232.95
	B	t/o	t/o

Table 1: Results for application to specification of m-floor elevator system.

7. RELATED WORK

Methods for computing locations of flaws in unrealizable specifications has been proposed in [4, 7]. Although in these methods they treat realizability, they restricted specification language to GR(1) which is a subset of LTL. In contrast to their works, we treat strong satisfiability instead of realizability, however, targets of our method are specifications written in full-LTL, and can be expanded to ω-regular languages easily.

Methods for computing minimal unsatisfiable subsets (MUSs) from unsatisfiable Boolean formulae are typically derived from research on satisfiability solvers, or SAT-solvers. In [9], procedures for computing all of the MUSs were proposed.

Methods for deriving causes (rather than the locations) of flaws in reactive system specifications have been reported [5, 8]. These methods derive constraints of the behavior of an environment implicitly imposed by unrealizable specifications. Because the implicit constraints provide hints to the cause of flaws in specifications, these constraints are helpful to correct the flaws.

8. CONCLUSION

We have described a method for computing MSUSs, using a counterexample, of a reactive system specification written

Events:

Environmental events:

$lbtn_i (i = 1..m)$, //Request button at ith floor is pushed.
$obtn$, //Open button in the lift is pushed.
$cbtn$ //Close button in the lift is pushed.

System events:

$l_i (i = 1..m)$, //The lift is located at ith floor.
$r_i (i = 1..m)$, //The lift is requested to go to ith floor.
opn, //The door is open.
mov, //The lift can move.
to, //The time-limit that the door can open has past.
$oreq$ //The door is requested to open.

Specifications:

(a) a specification for each floor

(af) functional requirements

//If a request button is pushed, the lift eventually go there.

$$\bigwedge_{1\le i\le m} \Box(lbtn_i \rightarrow \Diamond l_i \wedge r_i W (l_i \wedge r_i)),$$

//If the lift reaches the requested floor, the door open.

$$\bigwedge_{1\le i\le m} \Box(l_i \wedge r_i \rightarrow opn \wedge l_i W mov),$$

(an) non-functional requirements

//Until request button is pushed, lift is not requested to go there.

$$\bigwedge_{1\le i\le m} \Box(l_i \wedge mov \rightarrow (\neg r_i) W lbtn_i),$$

//If the lift is not requested at a floor, the door will not open there.

$$\bigwedge_{1\le i\le m} \Box(l_i \wedge \neg r_i \rightarrow \neg opn),$$

(c) a specification of the physical constraints

//The lift is located at some floor.

$$\Box(\bigvee_{1\le i\le m} l_i) \wedge \Box(\bigwedge_{1\le i\le m} (l_i \rightarrow \bigwedge_{j=1..m, i\neq j} \neg l_j)),$$

//The lift must pass floors on the way to the destination. ($m \geq 3$)

$$\Box(\bigwedge_{\substack{1\le i\le m-2 \\ 3\le j\le m, i\le j-2}} (l_i \wedge r_j \rightarrow \bigwedge_{i+2\le k\le j} (\neg l_k) U (\neg l_k \wedge l_{k-1}))),$$

$$\Box(\bigwedge_{\substack{1\le i\le m-2 \\ 3\le j\le m, i\le j-2}} (l_j \wedge r_i \rightarrow \bigwedge_{i+2\le k\le j} (\neg l_k) U (\neg l_k \wedge l_{k+1}))),$$

//A relation between the door open/close and the lift movable/unmovable

$\Box(opn \rightarrow (\neg mov) W (\neg opn))$,
$\Box(\neg opn \rightarrow mov W opn)$,

(b) a specification for a door of the lift

//The time-limit that the door open is set.
$\Box(opn \rightarrow \Diamond to)$,
//If open button is pushed, the door is requested to open.
$\Box(obtn \wedge \neg to \rightarrow oreq)$,
//When the time that door can open passed, the door closes.
$\Box(to \rightarrow \neg opn)$,
//If close button is pushed, the door closes.
$\Box(cbtn \wedge \neg oreq \rightarrow \neg opn)$,
//If the door is requested to open and the lift is not movable, the door opens.
$\Box(oreq \wedge \neg mov \rightarrow opn)$

(Remark: In the specifications, we use the weak until operator fWg, which is an abbreviation for $gR(g \vee f)$.)

Figure 3: Sample Specifications of m-floor elevator systems in [2].

in LTL which is not strongly satisfiable. Flaws can be located and corrected by modifying the parts of the specification indicated by the MSUSs. Because our method is based on strong satisfiability instead of realizability, it is possible to locate flaws in a specification efficiently. We proved correctness of our procedure, i.e., soundness and completeness, to ensure that all MSUSs are computed correctly. We also showed the time complexity of our procedure. Furthermore, we implemented the procedure, and confirmed that the set of MSUSs can be computed from a specification at a non-trivial scale in a reasonable amount of time.

STEP 6 simply translates minimal correction subsets (MCSs) into MUSs. In [9], several efficient translation algorithms are described. We expect that these algorithms can improve the efficiency of our method. We believe that our method has many potential practical applications for the analysis of safety-critical systems.

9. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number 24500032.

10. REFERENCES

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In *ICALP89*, vol. 372 of *LNCS*, pages 1–17. 1989.

[2] T. Aoshima and N. Yonezaki. Verification of reactive system specification with outer event conditional formula. In *International Symposium on Principles of Software Evolution (ISPSE2000)*, pages 195–199, 2000.

[3] T. Babiak, M. Kretínský, V. Rehák, and J. Strejcek. LTL to Büchi automata translation: Fast and more deterministic. In *TACAS2012*, vol. 7214 of *LNCS*, pages 95–109, 2012.

[4] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *VMCAI2008*, vol. 4905 of *LNCS*, pages 52–67. 2008.

[5] S. Hagihara, Y. Kitamura, M. Shimakawa, and N. Yonezaki. Extracting environmental constraints to make reactive system specifications realizable. In *16th Asia-Pacific Software Engineering Conference*, APSEC '09, pages 61–68, 2009.

[6] S. Hagihara and N. Yonezaki. Completeness of verification methods for approaching to realizable reactive specifications. In *AWCVS'06*, vol. 348 of *UNU-IIST Technical Report*, pages 242–257, 2006.

[7] R. Konighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD2009*, pages 152–159, 2009.

[8] W. Li, L. Dworkin, and S. A. Seshia. Mining assumptions for synthesis. In *MEMOCODE*, pages 43–50, 2011.

[9] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[10] R. Mori and N. Yonezaki. Several realizability concepts in reactive objects. In *Information Modeling and Knowledge Bases*, 1993.

[11] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL89*, pages 179–190, 1989.

[12] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.

[13] M. Shimakawa, S. Hagihara, and N. Yonezaki. Complexity of strong satisfiability problems for reactive system specifications. *IEICE Transactions on Information and Systems*, E96-D(10):2187–2193, 2013.

[14] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, K.-N. Wu, and W.-C. Chan. GOAL: A graphical tool for manipulating Büchi automata and temporal formulae. In *TACAS2007*, vol. 4424 of *LNCS*, pages 466–471. 2007.