Counterexample Classification

Cole Vick[1], Eunsuk Kang[2(✉)], and Stavros Tripakis[1(✉)]

[1] Northeastern University, Boston, USA
{vick.c,stavros}@northeastern.edu
[2] Carnegie Mellon University, Pittsburgh, USA
eskang@cmu.edu

Abstract. In model checking, when a given model fails to satisfy the desired specification, a typical model checker provides a *counterexample* that illustrates how the violation occurs. In general, there exist many diverse counterexamples that exhibit distinct violating behaviors, which the user may wish to examine before deciding how to repair the model. Unfortunately, obtaining this information is challenging in existing model checkers since (1) the number of counterexamples may be too large to enumerate one by one, and (2) many of these counterexamples are redundant, in that they describe the same type of violating behavior. In this paper, we propose a technique called *counterexample classification*. The goal of classification is to partition the space of all counterexamples into a finite set of *counterexample classes*, each of which describes a distinct type of violating behavior for the given specification. These classes are then presented as a summary of possible violating behaviors in the system, freeing the user from manually having to inspect or analyze numerous counterexamples to extract the same information. We have implemented a prototype of our technique on top of an existing formal modeling and verification tool, the Alloy Analyzer, and evaluated the effectiveness of the technique on case studies involving the well-known Needham-Schroeder protocol with promising results.

1 Introduction

In formal verification, *counterexamples* are an invaluable aid for debugging a system model for possible defects. Typically, a counterexample is constructed by a verification tool as a *trace* (i.e., a sequence of states or events) that demonstrates how the system violates a desired property. The user of the tool would then inspect the counterexample for the underlying cause behind the violation and fix the model accordingly.

In practice, there are a number of challenges that the user may encounter while using counterexamples to debug and repair a model. First, a counterexample may contain details that are irrelevant to the root cause of a violation, requiring considerable effort by the user to manually analyze and extract the violating behavior. Second, the user may wish to investigate multiple different

types of counterexamples before deciding how to repair the model; this is, however, a challenging task because (1) the number of counterexamples may be too large to enumerate one by one, and (2) many of these counterexamples may be *redundant* in that they describe the same type of violating behavior.

This paper proposes a technique called *counterexample classification* as an approach to overcome these challenges. The key intuition behind this approach is that although a typical model contains a very large (or possibly infinite) set of counterexamples, (1) many of these can be considered "similar", in that they share a common, violating behavior and (2) this similarity can be captured as a specific relationship between states that is shared by these traces. Based on this insight, our technique automatically partitions the set of counterexamples into a finite number of *classes*, each of which is associated with a *constraint* that characterizes one particular type of violation. These constraints are then presented to the user, along with representative counterexamples, as distinct descriptions of possible defects in the system, freeing them from manually sorting through numerous counterexamples to extract the same information.

We have built a prototype implementation of our classification technique on top of an existing formal modeling and verification tool, the Alloy Analyzer [8]. Our tool accepts a formal model, a specification (that the model currently violates), and a set of *predicates* that describe relationships between states in the model. From these, the tool produces (if one exists) a set of classes that accounts for all of the violating behavior in the model. As a case study, we have successfully applied our technique to two variants of the Needham-Schroeder protocol [12], and were able to classify hundreds of thousands of counterexamples into only a handful of classes that represent known attacks to the protocol.

Our main contributions may be summarized as follows: a formal definition of the Counterexample Classification Problem (Sect. 3), a solution to the Counterexample Classification Problem (Sect. 4), and a case study on a well-established distributed protocol, Needham-Schroeder (Sect. 5), that demonstrates the efficacy of our solution.

1.1 Running Example

To motivate our technique, we introduce the following example. `Alice` and `Bob` are sending `Message`s to each other. `Eve` is able to view these messages as they are being sent. The content of a message can be either `Plaintext` or `Encrypted`. `Eve` is always able to read `Plaintext` messages, but needs Key_{AB}, Alice and Bob's shared key, to read `Encrypted` messages. `Eve` acquires Key_{AB} by seeing an `Encrypted` message, modelling `Eve` "breaking" the encryption of what should be a one-time key. A `Message` may be flagged as `Secret`, meaning that its content should not be read by `Eve`.

We model this example as a transition system, shown in Fig. 1. The transition system has four states, represented by two state variables, $EveKey$ of type $Key = \{\emptyset, \text{Key}_{\text{AB}}\}$, and $EveSeenSecret$ of type Boolean ($\top$ for true and $\bot$ for false). The initial state is $(\emptyset, \bot)$ meaning that `Eve` does not know the key and has not read any secret.
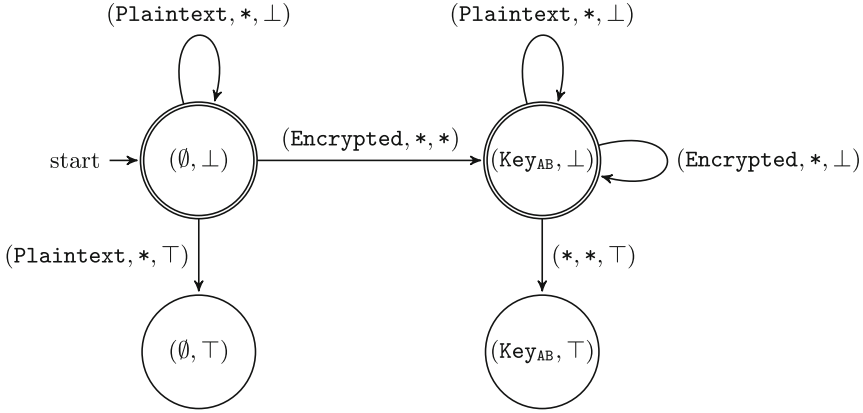
Fig. 1. Transition system of the running example.

Transitions between states are labeled by `Message`s. A `Message` is a tuple of the form $(type, sender, secret)$, where $type \in \{\texttt{Encrypted}, \texttt{Plaintext}\}$ denotes whether the message is encrypted or not (if encrypted, a message is encrypted by $\texttt{Key}_{\texttt{AB}}$), $sender \in \{\texttt{Alice}, \texttt{Bob}\}$ denotes the sender of the message, and $secret$ is a Boolean denoting whether the message is secret or not. For example, the transition $(\emptyset, \bot) \xrightarrow{(\texttt{Plaintext}, \texttt{Alice}, \top)} (\emptyset, \top)$ means that `Alice` sends a `Plaintext` (unencrypted) `Secret` message.[1] $*$ indicates that the corresponding field can take any value within its type, i.e., there are multiple such transitions, one for each possible value.

We would like this system to satisfy the property that `Eve` never reads a `Message` that is flagged as `Secret`. This can be expressed as the temporal logic (LTL) formula

$$\Phi = \mathbf{G}(\mathit{EveSeenSecret} = \bot)$$

which states that $\mathit{EveSeenSecret} = \bot$ holds at every reachable state of the system, i.e., it is an *invariant*. As we can see, this is not the case for the model in Fig. 1. The top two double bordered states are the *good* states.

Note that this system has infinitely many *counterexample traces*, as self-loop transitions can be taken arbitrarily many times.

Take for instance the counterexample traces listed below:

$$\rho_1^1 = (\emptyset, \bot) \xrightarrow{(\texttt{Plaintext}, \texttt{Alice}, \top)} (\emptyset, \top)$$

$$\rho_1^2 = (\emptyset, \bot) \xrightarrow{(\texttt{Plaintext}, \texttt{Alice}, \bot)} (\emptyset, \bot) \xrightarrow{(\texttt{Plaintext}, \texttt{Alice}, \top)} (\emptyset, \top)$$

$$\rho_1^3 = (\emptyset, \bot) \xrightarrow{(\texttt{Plaintext}, \texttt{Bob}, \bot)} (\emptyset, \bot) \xrightarrow{(\texttt{Plaintext}, \texttt{Bob}, \top)} (\emptyset, \top)$$

$$\rho_1^4 = (\emptyset, \bot) \xrightarrow{(\texttt{Plaintext}, \texttt{Alice}, \bot)} (\emptyset, \bot) \xrightarrow{(\texttt{Plaintext}, \texttt{Bob}, \top)} (\emptyset, \top)$$

---

[1] The traces in this section have labels, i.e. `Message`s, on their transitions. We do this to make it clear how messages are sent and how different messages affect the state. Our formal definition will not include labels as they may be encoded directly into the state.

In ρ_1^1, `Alice` sends a `Plaintext Secret` message. `Eve` is be able to read it, as it is unencrypted, which leads to a violation of the property. In ρ_1^2, `Alice` first sends a `Plaintext` but non-secret message and then sends a `Plaintext Secret` message. In ρ_1^3, `Bob` first sends a `Plaintext` but non-secret message and then he sends a `Plaintext Secret` message. In ρ_1^4, `Alice` sends a `Plaintext` but non-secret message and then `Bob` sends a `Plaintext Secret` message.

These violating traces share important behavior: the fact that either `Alice` or `Bob` sends a `Plaintext Secret` message. Noticing this, we would like to group these traces together in the same *counterexample class*.

Now consider the counterexample traces listed below:

$$\rho_2^1 = (\emptyset, \bot) \xrightarrow{(\texttt{Encrypted,Alice},\bot)} (\texttt{Key}_{\texttt{AB}}, \bot) \xrightarrow{(\texttt{Encrypted,Bob},\top)} (\texttt{Key}_{\texttt{AB}}, \top)$$

$$\rho_2^2 = (\emptyset, \bot) \xrightarrow{(\texttt{Encrypted,Bob},\bot)} (\texttt{Key}_{\texttt{AB}}, \bot) \xrightarrow{(\texttt{Encrypted,Alice},\top)} (\texttt{Key}_{\texttt{AB}}, \top)$$

$$\rho_2^3 = (\emptyset, \bot) \xrightarrow{(\texttt{Encrypted,Alice},\top)} (\texttt{Key}_{\texttt{AB}}, \bot) \xrightarrow{(\texttt{Encrypted,Alice},\top)} (\texttt{Key}_{\texttt{AB}}, \top)$$

$$\rho_2^4 = (\emptyset, \bot) \xrightarrow{(\texttt{Encrypted,Bob},\top)} (\texttt{Key}_{\texttt{AB}}, \bot) \xrightarrow{(\texttt{Encrypted,Alice},\top)} (\texttt{Key}_{\texttt{AB}}, \top)$$

These traces exhibit a different way in which the property can be violated than the traces shown previously. Now, the violation happens when `Alice` or `Bob` send an `Encrypted Secret` message after an `Encrypted` message has already been sent, i.e. after `Eve` has broken the encryption. A description of this new class would be: `Eve` receives an `Encrypted` message before receiving an `Encrypted Secret` message.

The method and tool presented in this paper generate such counterexample classes automatically. Our tool does not output class descriptions in English but represents classes syntactically as *trace constraints*. A trace constraint is evaluated over a given trace ρ. If ρ satisfies the trace constraint then we say that ρ falls into the class that the trace constraint represents. The trace constraints that represent the two classes discussed above are:

$$TC_{Plaintext}[\rho] \equiv \exists i \in [0..len(\rho)] : \rho.type@i = \texttt{Plaintext} \land \rho.secret@i = \top$$
$$TC_{Encrypted}[\rho] \equiv \exists i, j \in [0..len(\rho)] : i < j \land \rho.EveKey@i = \texttt{Key}_{\texttt{AB}} \land$$
$$\rho.type@j = \texttt{Encrypted} \land \rho.secret@j = \top$$

where $len(\rho)$ denotes the length of trace ρ and the variables i and j represent indices to particular positions of states and transitions in ρ. The initial state is indexed at position s_0 and the first transition is indexed at position l_0 and leads to state s_1 thus following the general pattern: $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \cdots$.

2   Background

Definition 1 (Symbolic transition system). *A symbolic transition system is a tuple (X, I, T) where:*

– *X is a finite set of typed state variables. Each variable $x \in X$ has a type, denoted $\mathsf{type}(x)$. A type is a set of values.*

– *The* initial state predicate I *is a predicate (i.e., Boolean expression) over X.*
– *The* transition relation predicate T *is a predicate over $X \cup X'$, where X' denotes the set of* primed *(next state) variables obtained from X. For example, if $X = \{x, y, z\}$ then $X' = \{x', y', z'\}$. Implicitly, every primed variable has the same type as the original variable: $\forall x \in X : \mathsf{type}(x') = \mathsf{type}(x)$.*

We let U denote the universe of all values. A *state* s over a set of state variables X is an assignment of a value (of the appropriate type) to each variable in X, i.e., s is a (total) function $s : X \to U$, such that $\forall x \in X : s(x) \in \mathsf{type}(x)$. A state s satisfies a predicate I over X, denoted $s \models I$, if when we replace all variables in I by their values as defined by s, I evaluates to true. For example, suppose $X = \{x, y, z\}$ where x and y are integer variables, and z is a Boolean variable. Let I be the predicate $x < y \wedge z$. Consider two states, $s_1 = (x = 3, y = 4, z = \top)$ and $s_2 = (x = 3, y = 1, z = \top)$. Then, $s_1 \models I$ but $s_2 \not\models I$.

Similarly, a pair of states (s, s') satisfies a predicate T over $X \cup X'$ if when we replace all variables from X in T by their values as defined by s, and all variables from X' in T by their values as defined by s', T evaluates to true. For example, suppose $X = \{x\}$ where x is an integer variable. Let T be the predicate $x' = x + 1$. Consider three states, $s_0 = (x = 0)$, $s_1 = (x = 1)$, and $s_2 = (x = 2)$. Then $(s_0, s_1) \models T$ and $(s_1, s_2) \models T$, but $(s_0, s_2) \not\models T$.

Definition 2 (Transition system defined from a symbolic transition system). *A symbolic transition system (X, I, T) defines a transition system (S, S_0, R), where:*

– *The set of* states S *is the set of all assignments over X.*
– *The set of* initial states S_0 *is the set: $S_0 = \{s \in S \mid s \models I\}$.*
– *The* transition relation R *is the set: $R = \{(s, s') \in S \times S \mid (s, s') \models T\}$.*

That is, the set of initial states is the set of all states satisfying I, and the transition relation R is the set of all pairs of states satisfying T. A pair $(s, s') \in R$ is also called a *transition*, and is sometimes denoted $s \to s'$.

Definition 3 (Trace). *A trace ρ over a set of state variables X is a finite sequence of states over X: $\rho = s_0, ..., s_k$. The length of ρ is k, and is denoted by $\mathsf{len}(\rho)$; note that k may equal 0, in which case the trace is empty. The set of states of ρ is $\{s_0, ..., s_k\}$ and is denoted $\mathsf{States}(\rho)$.*

Definition 4 (Property). *A property Φ over a set of state variables X is a set of traces over X.*

Definition 5 (Traces for an STS). *Let $\mathsf{STS} = (X, I, T)$ be a symbolic transition system and let (S, S_0, R) be the transition system of STS. The set of traces generated by STS, denoted $\mathsf{Traces}(\mathsf{STS})$, is the set of all traces $\rho = s_0, s_1, ..., s_k$ over X such that:*

– *$s_0 \in S_0$. That is, ρ starts at an initial state of STS.*
– *$\forall i \in \{0, ..., k - 1\} : (s_i, s_{i+1}) \in R$. That is, every pair of successive states in ρ is linked by a transition in STS.*

Definition 6 (Property satisfaction and counterexamples). *Let* $\mathsf{STS} = (X, I, T)$ *be a symbolic transition system and let Φ be a property over X. We say that* STS *satisfies* Φ, *written* $\mathsf{STS} \models \Phi$, *iff* $\mathsf{Traces}(\mathsf{STS}) \subseteq \Phi$. *If* $\mathsf{STS} \not\models \Phi$, *then a* counterexample *is any trace* $\rho \in \mathsf{Traces}(\mathsf{STS}) \setminus \Phi$, *i.e., any trace of* STS *which violates (does not belong in) Φ.*

3   Counterexample Classification

3.1   Classes and Classifications

Consider a set of traces P. A *class of* P is any non-empty subset of P. A *classification of* P is a partition of P into (not necessarily disjoint) classes.

Definition 7 (Classification). *Consider a set of traces P. A* classification of P *is a finite set C of classes of P such that* $\bigcup_{c \in C} c = P$.

Given a set of counterexample traces P, and a classification C of P, a *canonical counterexample* is a counterexample trace that belongs in exactly one class of C. A canonical counterexample thus represents the violating behavior of a particular class as it only appears in that particular class.

Definition 8 (Canonical Counterexample). *Given a set of counterexamples traces P and a classification C of P, a* canonical counterexample ρ *is any counterexample in P such that:* $\forall c_1, c_2 \in C : (\rho \in c_1 \land \rho \in c_2) \rightarrow c_1 = c_2$. *We denote by $c(\rho)$ the unique class in C that ρ belongs to.*

A classification is *redundant* if it contains classes that have no canonical counterexample:

Definition 9 (Redundant Classification). *A classification C of a set of counterexamples P is* redundant *if there exists a class $c \in C$ such that c does not contain a canonical counterexample.*

Example 1. Suppose $P = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$ and $C = \{c_1, c_2, c_3\}$ with $c_1 = \{\rho_1, \rho_2, \rho_3\}, c_2 = \{\rho_3, \rho_4, \rho_5\}, c_3 = \{\rho_1, \rho_4\}$. Note that C is a valid classification of P as $c_1 \cup c_2 \cup c_3 = P$. C is a redundant classification, because although c_1 has a canonical counterexample ρ_2, and c_2 has canonical counterexample ρ_4, c_3 has no canonical counterexample.

Often, we would like for a classification to guarantee that each class has a canonical counterexample, i.e., to be *non-redundant*. In general, we can transform every redundant classification into a non-redundant classification. First, we state the following two lemmas[2]:

Lemma 1. *A classification C of a set of counterexamples P is redundant iff there exist distinct classes $c, c_1, ..., c_n \in C$ such that $c \subseteq \bigcup_{i=1,...,n} c_i$.*

---

[2] Proofs for the following Lemmas and Theorems have been removed due to page restrictions. The full paper, with proofs, is available here https://arxiv.org/abs/2108.00885.

Lemma 2. *Let $C = \{c_1, ..., c_n\}$ be a classification of a set of counterexamples P. C is redundant iff there exists $i \in \{1, ..., n\}$ such that $c_i \subseteq \bigcup_{j \neq i} c_j$.*

Based on Lemma 2, we can construct an algorithm to transform any classification into a non-redundant classification.

3.2   The Counterexample Classification Problem

In Sect. 3.1, we defined the concepts of classes and classifications *semantically*. But in order to define the counterexample classification problem that we solve in this paper, we need a *syntactic* representation of classes. We define such a representation in this section, by means of *trace constraints*. A trace constraint is a special kind of predicate that evaluates over traces. A trace constraint is similar to predicates such as the I (initial state) predicate of a symbolic transition system, with two key differences: (1) a trace constraint is only conjunctive, and (2) a trace constraint can refer to state variables at certain positions in the trace and impose logical conditions over those positions. For example, if $X = \{x, y\}$ is the set of state variables, then here are some examples of trace constraints:

- $TC1[\rho] \equiv \exists i \in [0..len(\rho)] : x@i = y@i$: this trace constraint says that there is a position i in the trace such that the value of x at that position is the same as the value of y.
- $TC2[\rho] \equiv \exists i, j \in [0..len(\rho)] : i < j \wedge x@i > x@j$: this says that there are two positions i and j in the trace such that i is earlier than j and the value of x decreases from i to j.

We call formulas such as $x@i = y@i$ or $x@i > x@j$, which operate on indexed state variables, *atomic facts*. We call formulas such as $i < j$, which operate on position variables, *atomic position facts*. Then, a trace constraint is a conjunction of atomic facts and atomic position facts, together with an existential quantification of all position variables within the range of the length of the trace.

Atomic facts and atomic position facts are defined over a set of *user-defined predicates*. Some predicates will be standard, such as *equality* ($=$) for integers and *less-than* ($<$) for positions, while other predicates may be domain-specific. In addition to variables, we allow predicates to refer to constants. For example, $i \leq 10$ says that the position i must be at most 10, and $x@2 = 13$ says that the value of x at position 2 must be 13.

For example, recall the `Message` type from the running example. The user might want to define a predicate that checks whether two messages have the same sender. Then, the user can define the predicate *SendersEqual* which is parameterized over two variables of type `Message` and defined as:

$$SendersEqual[m_1, m_2] \equiv m_1.sender = m_2.sender$$

This predicate may be then instantiated as:

$$SendersEqual[message@1, message@5]$$

This checks whether the `Message` at position 1 has the same sender as the `Message` at position 5.

Definition 10 (Trace Constraint). *A trace constraint over a set of state variables X and a set V of user-defined predicates is a formula of the form*

$$TC[\rho] \equiv \exists i_1, ..., i_k \in [0..len(\rho)] : \xi_0 \wedge \xi_1 \wedge \cdots \wedge \xi_n$$

where:

- *$i_1, ..., i_k$ are non-negative integer variables denoting positions in the trace t. We allow k to be 0, in which case the trace constraint has no position variables.*
- *Each ξ_j, for $j = 0, ..., n$, is either an atomic fact over state variables X and position variables $i_1, ..., i_k$ or an atomic position fact over position variables $i_1, ..., i_k$ using predicates in V.*

Given a trace constraint w, and a trace ρ, we can evaluate w on ρ in the expected way. For example, the trace $(x = 0) \longrightarrow (x = 0)$ over state variable x, satisfies the trace constraint $TC_1[\rho] \equiv \exists i_0, i_1 \in [0..len(\rho)] : i_0 < i_1 \wedge x@0 = x@1$ but does not satisfy the trace constraint $TC_2[\rho] \equiv \exists i_0, i_1 \in [0..len(\rho)] : i_0 < i_1 \wedge x@0 > x@1$. We write $\rho \models w$ if trace ρ satisfies trace constraint w. We also say that w *characterizes* ρ when $\rho \models w$. We denote by $c(w)$ the set of all traces satisfying constraint w.

Let W be a set of trace constraints. Then, let $C(W) = \{c(w) \mid w \in W\}$; i.e., $C(W)$ is the set of all sets of traces that are characterized by some trace constraint in W.

Consider a symbolic transition system STS and a property Φ that is violated by STS, i.e., $\mathsf{STS} \not\models \Phi$. The problem that we are concerned with in this paper is to find a classification of all traces of STS that violate Φ, such that this classification is represented by a set of trace constraints defined over V. We call this problem the *counterexample classification problem* (CCP):

Definition 11 (Counterexample Classification Problem). *Given symbolic transition system $\mathsf{STS} = (X, I, T)$, property Φ such that $\mathsf{STS} \not\models \Phi$, and user-defined predicates V, find, if there exists, a set of trace constraints W such that: (1) each $w \in W$ is a trace constraint over X and V; and (2) $C(W)$ is a classification of P, where P is the set of all traces of STS that violate Φ.*

Lemma 3. *Let W be a solution to the CCP. Then, every trace constraint $w \in W$ is a sufficient condition for a violation, i.e., $\forall w \in W : c(w) \cap \Phi = \emptyset$.*

3.3 Solvability

The CCP is formulated as to find a set of trace constraints W *if one exists* (Definition 11). Indeed, while a semantic classification always exists (e.g., a trivial one is the one containing just one class, the set of all counterexamples P), a *syntactic classification* in the form of W might not always exist. Whether or not one exists depends on the set of user-defined predicates V.

Lemma 4. *If the set of counterexample traces P is finite, and V includes equality $=$, then CCP always has a solution.*

Lemma 4 shows that in the presence of equality $=$, and provided that the set of counterexamples is finite, CCP always has a solution. But in the absence of $=$, CCP may not have a solution.

For example, consider an STS with $X = \{a\}$ where a is an integer variable that can be non-deterministically incremented by 1, decremented by 1, or held constant at each step. Let the initial state be $a = 1$. Let the property Φ be $\mathbf{G}(a = 1)$, i.e., we require that a is always 1, which is clearly violated by this system.

Suppose that V only contains the predicate $lessThanOne[x]$, which returns true if and only if the given integer x is strictly less than 1. Then, we claim that CCP has no solution. Indeed, note that the set of counterexample traces includes all traces where at some point either $a < 1$ or $a > 1$. But the given V is unable to generate an atomic fact where a is greater than 1 (notice that negation is not allowed in trace constraints). Therefore we cannot classify all counterexample traces, and in particular not those where $a > 1$.

Now suppose that we change V to $\{lessThanOne, greaterThanOne\}$, with the obvious meanings. Then the following two trace constraints constitute a solution to CCP:

$$TC_1[\rho] \equiv \exists i \in [0..len(\rho)] : lessThanOne[x@i]$$
$$TC_2[\rho] \equiv \exists i \in [0..len(\rho)] : greaterThanOne[x@i]$$

3.4   Uniqueness of Solutions

The discussion in Sect. 3.3 shows that CCP may or may not have a solution, depending on the set V of predicates allowed in the trace constraints. In this section we show that even for a fixed V, CCP does not necessarily have a unique solution.

Consider the example given just above, in Subsect. 3.3. If we set V to $\{lessThanOne, greaterThanOne, \neq\}$, where $\neq$ is the not-equals predicate, the problem now admits at least two solutions. W_1 is still a solution, while the second solution $W_2 = \{TC_3\}$ uses only the $\neq$ predicate to characterize the violating behavior. The trace constraint TC_3 is defined as:

$$TC_3[\rho] \equiv \exists i \in [0..len(\rho)] : x@i \neq 1$$

4   Classification Method

In this section, we present a method for solving the CCP introduced in Sect. 3.2. We present an overview of our proposed classification algorithm (Sect. 4.1), describe optimizations to ensure the generation of a non-redundant classification with minimal classes (Sect. 4.2), and finally present a solution to the Running example (Sect. 4.3).

Input : An STS, a specification Φ, and a set of predicates V
Output: A set of trace constraints W
1 Func classify(STS, Φ, V)*:*
2 | $W = \emptyset$
3 | **while** verify(STS $\wedge$ block(W), Φ) $==$ Violated **do**
4 | | ρ = counterexample(STS $\wedge$ block(W), Φ)
5 | | Γ = facts(ρ, V)
6 | | **if** $\Gamma = \emptyset$ **then**
7 | | | **return** *"V cannot sufficiently characterize the violation in ρ"*
8 | | w = traceConstraint(Γ, ρ)
9 | | **if** verify(STS $\wedge$ w, $\neg\Phi$) $==$ Violated **then**
10 | | | **return** *"V cannot sufficiently characterize the violation in ρ"*
11 | | w = minimizeTC(STS, w, Φ)
12 | | $W = W \cup w$
13 | W = removeRedundant(STS, W, Φ)
14 | **return** W

Algorithm 1: The counterexample classification algorithm.

4.1   Algorithm Overview

Given an STS, a property Φ, and a set of user-defined predicates V, the goal is to find a set of trace constraints W such that $C(W)$ is a solution to the CCP (Definition 11 in Sect. 3.2). We assume, without loss of generality, that V is non-empty. Indeed, an empty V implies that the only possible trace constraint is the empty trace constraint, which characterizes the set of all traces. This situation can be modelled by adding to V a trivial predicate that always returns $\top$ (true), thus having a non-empty V. To guarantee termination, we assume that the set of counterexamples $P = \text{Traces}(\text{STS}) \setminus \Phi$ is finite. To avoid the trivial solution where all traces are violating, we also assume that $\text{Traces}(\text{STS}) \cap \Phi \neq \emptyset$.

The pseudocode for the classification algorithm is shown in Algorithm 1. Procedure classify relies on the existence of a *verifier* that is capable of checking STS against Φ and generating a counterexample trace, if it exists. In particular, classify uses the following verifier functions:

– verify(STS $\wedge$ φ, Φ): Returns OK if STS satisfies Φ under the additional constraint φ, i.e., if $\text{Traces}(\text{STS} \wedge \varphi) \subseteq \Phi$; else, returns Violated. The constraint φ is typically a trace constraint. We provide examples of φ later in this section.
– counterexample(STS $\wedge$ φ, Φ): If verify(STS $\wedge$ φ, Φ) $==$ Violated, returns a trace ρ of STS such that $\rho \models \varphi$ and $\rho \not\models \Phi$; else, returns an empty output.

The algorithm begins by checking whether STS violates Φ (line 3) and if so, returning a counterexample that demonstrates how a violation can occur (line 4). The additional argument to the verifier, block(W), is used to prevent the verifier from re-generating a counterexample that belongs to any previously generated classes; we will describe this in more detail later in this section.

Next, given a particular counterexample ρ, the helper function facts generates the set Γ of all atomic facts and atomic position facts that hold over ρ, by instantiating the predicates V over the states in ρ (line 5). Then, based on Γ,

traceConstraint builds a trace constraint that characterizes ρ. In particular, this procedure transforms Γ into a syntactically valid trace constraint w, by (1) introducing a sequence of existential quantifiers over all positional variables in ρ and (2) taking the conjunction of all facts in Γ (line 8).

In the next step, the verifier is used once again to ensure that the trace constraint w sufficiently captures the violating behavior in ρ (line 9). This is done by checking that every trace of STS that satisfies w (i.e., it shares the same characteristics of ρ as described by Γ) results in a violation of Φ. If not, it implies that w is not strong enough to guarantee a violation; i.e., V does not contain enough predicates to fully characterize ρ. In this case, a solution to the CCP cannot be produced and the algorithm terminates with an error (line 10).

If w guarantees a violation, it is added to the set of classes that will eventually form a solution classification to the CCP (line 12). The process from lines 4 to 12 is then repeated until it exhausts the set of all counterexample classes for STS and Φ.

To prevent the verifier from returning the same type of counterexample as ρ, classify passes block(W) as an additional constraint to verify, where:

$$\mathsf{block}(W) \equiv \neg(\bigvee_{i=1}^{|W|} w_i)$$

In other words, by including block(W) as an additional constraint, the verifier ensures that it only explores traces that do not belong to any of the classes in W. Note that if W is empty (as in the first iteration of the loop), block(W) returns *true* (i.e., $\top$).

Once the verifier is no longer able to find any counterexample, the algorithm terminates by returning W as the solution classification (line 14).

Provided there is a finite number of counterexamples and a non-empty set of accepting traces, Algorithm 1 terminates because at least one counterexample is classified at each iteration of the while loop. The following theorems establish the correctness of the algorithm.

Theorem 1. *Any W returned by* classify *is a valid solution to the CCP.*

Theorem 2. *If* classify *returns no solution (lines 7 or 10 of Algorithm 1), then CCP has no solution for the given V.*

Example 2. Recall the example from Sect. 3.3. To make P finite, we assume that the length of counterexample traces is exactly 2. Then, $P = \{(a = 1) \xrightarrow{--} (a = 0), (a = 1) \xrightarrow{++} (a = 2)\}$. Let the set of user-defined predicates be $V = \{lessThanOne, greaterThanOne\}$.

Suppose that the verifier returns $\rho = (a = 1) \xrightarrow{--} (a = 0)$ as the first counterexample (line 4). Next, facts evaluates the predicates in V over the state variable a at position 0 and 1 (line 5), producing Γ that contains one fact: $\{lessThanOne[a@1]\}$. Then, the trace constraint w constructed based on Γ is:

$$TC_1[\rho] \equiv \exists i_1 \in [0..len(\rho)] : lessThanOne[a@i_1]$$

It can be shown that any trace of STS that satisfies TC_1 is a violation of Φ; thus, this newly created constraint $w \equiv TC_1$ is added to the set W.

In our example, there is one more counterexample; namely, $\rho = (a = 1) \xrightarrow{++} (a = 2)$, which can be used to construct the following additional trace constraint:

$$TC_2[\rho] = \exists i_1 \in [0..len(\rho)] : greaterThanOne[a@i_1]$$

Once TC_2 is added to W, there are no more remaining counterexamples, and the algorithm terminates by returning $W = \{TC_1, TC_2\}$.

4.2   Optimizations

Minimizing Trace Constraints. A trace constraint w generated on line 6 in Algorithm 1 may be a sufficient characterization of ρ, but it may also contain facts that are *irrelevant* to the violation. To be more precise, we consider a fact $f \in \Gamma$ to be irrelevant if trace constraint w that is constructed from $\Gamma' \equiv \Gamma - f$ is still sufficient to imply a violation.

Let us revisit Example 2. Suppose that we add to the set V of user-defined predicates an additional predicate $<$ over position variables. Then, for the counterexample $\rho = (a = 1) \xrightarrow{--} (a = 0)$, facts returns $\Gamma = \{lessThanOne[a@1], 1 < 2\}$ where 1 and 2 are positions in ρ. Then, the trace constraint generated by traceConstraint will be:

$$TC_3[\rho] = \exists i_1, i_2 \in [0..len(\rho)] : lessThanOne[a@i_2] \wedge i_1 < i_2$$

Although TC_3 is sufficient to imply a violation, it is less general than the previously generated TC_1 in the absence of predicate $<$ (see Example 2). Indeed, the constraint $i_1 < i_2$ in TC_3 forces the condition $a < 1$ to occur only at positions $i_2 > 0$, whereas in TC_1 the same condition can also occur at position $i_1 = 0$. Furthermore, this additional constraint can be safely removed from TC_3 while still guaranteeing a violation. Thus, constraint $i_1 < i_2$ is an irrelevant fact.

Our algorithm performs an additional *minimization* step to remove all such irrelevant facts from w. This additional procedure provides two benefits: (1) it reduces the amount of information that the user needs to examine to understand the classes and (2) each minimized class is a generalization of the original class and covers an equal or larger set of traces that share the common characteristics, thus also reducing the number of classes in the final classification.

As shown in Algorithm 2, minimizeTC relies on the ability of certain verifiers (such as the ones based on SAT [8] or SMT solvers [4]) to produce a *minimal core* for the unsatisfiability of a formula [15]. In particular, minCore(STS, w, $\neg\Phi$) computes a minimal subset of conjuncts in the symbolic representation of STS and w that are sufficient to ensure that $\neg\Phi$ holds (line 6). The facts (γ) that are common to this core and Γ represent the minimal subset of facts about ρ that are sufficient to imply a violation; a new trace constraint is then constructed based on this subset and returned as the output of minimizeTC (line 7).

Note that if verify on line 4 returns Violated (i.e., $\neg\Phi$ does not always hold under constraint w), this implies that the set of facts in Γ is not sufficient

Input   : An STS, a trace constraint w, and a specification Φ
Output: A minimized trace constraint

1 **Func** minimizeTC(STS, w, Φ):
2   **if** verify(STS $\wedge w$), $\neg\Phi$) $==$ OK **then**
3         $\gamma = \Gamma \cap$ minCore(STS, $w, \neg\Phi$)
4   **return** traceConstraint(γ, ρ)
5   **else**
6   **return** *"Γ does not sufficiently characterize the violation in ρ"*

Algorithm 2: minimizeTC, which removes from trace constraint w all facts that are irrelevant to the violation depicted by ρ.

to imply a violation of Φ. However, if minimizeTC is invoked from line 9 in Algorithm 1, this side of the conditional branch should never be reachable.

Non-Redundancy. Although non-redundancy of classification W is not necessary for a valid solution to the CCP, it is a desirable property as it reduces the number of classes that the user needs to inspect. Thus, the main algorithm classify also performs a redundancy check at its end (line 11, Algorithm 1) to ensure the non-redundancy of any solution that it produces.

Input   : an STS, a set of trace constraints W, and a specification Φ
Output: a set of trace constraints W'

1 **func** removeRedundant(STS, W, Φ):
2     $W' = \emptyset$
3   **for** $w \in W$ **do**
4   **if** verify(STS $\wedge$ block($W \setminus \{w\}$), Φ) $==$ Violated **then**
5             $W' = W' \cup w$
6   **return** W'

Algorithm 3: removeRedundant checks whether any $w \in W$ is redundant and if it is, removes it.

Function removeRedundant, shown in Algorithm 3, ensures that no trace constraint $w \in W$ is covered by any other trace constraints in W. Note that when the while loop in Algorithm 1 is exited, verify(STS $\wedge$ block(W), Φ) returns OK since W classifies all counterexamples in P. This means that all traces of STS which do not belong in any of the classes in W satisfy Φ. To find redundant trace constraints, we iterate over each $w \in W$ and check whether STS still satisfies Φ with w removed from W (line 4, Algorithm 3). If this is the case, then w is redundant, since $W \setminus \{w\}$ already covers P. Otherwise, w must characterize some $\rho \in P$ that the other trace constraints do not, and thus w is added to the non-redundant set W', which is returned at the end.

For example, recall the predicates $V = \{\neq, lessThanOne, greaterThanOne\}$ from Sect. 3.4. Suppose that classify finds two trace constraints in this order[3]:

$$TC_1[\rho] \equiv \exists i \in [0..len(\rho)] : lessThanOne[a@i]$$
$$TC_2[\rho] \equiv \exists i \in [0..len(\rho)] : a@i \neq 1$$

---

[3] Note that a newly created trace constraint is never redundant.

Notice that TC_2 classifies all counterexamples that TC_1 classifies. Thus, TC_1 is redundant and is not added to the final solution $W' = \{TC_2\}$.

4.3 Solution to the Running Example

Consider the running example presented in Sect. 1.1. For this example, Algorithm 1 outputs the trace constraints $TC_{Encrypted}$ and $TC_{Plaintext}$ in Sect. 1.1 given the set of predicates $V = \{=, <\}$. Equality $=$ operates over Messages and Booleans while $<$ operates on position variables.

Atomic position facts are generated just like atomic facts. Recall the following counterexample trace that is characterized by $TC_{Encrypted}$:

$$\rho = (\emptyset, \bot) \overset{(\texttt{Encrypted,Alice}, \bot)}{\longrightarrow} (\texttt{Key}_{\texttt{AB}}, \bot) \overset{(\texttt{Encrypted,Alice}, \top)}{\longrightarrow} (\texttt{Key}_{\texttt{AB}}, \top)$$

In the facts procedure, the $<$ predicate would generate two facts, $\{i_1 < i_2, i_2 < i_3\}$. These facts impose an ordering on any satisfying counterexample and capture the timing of the violation.

5 Implementation and Case Studies

5.1 Implementation

We have built a prototype implementation of the classify algorithm (Algorithm 1) on top of the Alloy Analyzer [8], a formal modeling and verification tool. In particular, Alloy uses an off-the-shelf SAT solver to perform bounded model checking (BMC), which is used for the verify procedure in the algorithm. As we demonstrate in this section, our prototype is capable of characterizing a large set of counterexamples (hundreds of thousands) with only a handful of generated classes. These generated classes are provided to the user in the form of trace constraints, along with representative counterexamples from each class.

Even though our current implementation uses Alloy and BMC, our technique does not depend on the use of BMC or any particular verification engine and could be implemented using other tools, provided they are capable of generating counterexample traces. Our current implementation does rely on the SAT solver being able to compute minimal unsatisfiable cores (which are used for minimizing the trace constraints).

5.2 Case Studies: Needham-Schroeder

As a case study, we applied our prototype to the well-known Needham-Schroeder protocol (NSP) [12], which has been known to be vulnerable to certain types of attacks [11]. We show how our classification methods can be used to classify the large number of counterexamples in a formal model of NSP into a small number of classes that correspond to these types of attacks.

The purpose of NSP is to allow two parties to communicate privately over an insecure network. NSP has two variants that look to accomplish this goal in

different ways. The first variant is the Needham-Schroeder Symmetric protocol, from now on referred to as Symmetric, and the second variant is the Needham-Schroeder Public-Key protocol, from now on referred to as Public-Key. The two variants exhibit different violating behaviors, which allowed us to test our classification technique on the two separate variants, while not having to write two drastically different models.

Formal Modeling. We constructed Alloy models of both the Symmetric and Public-Key variants. Together, both variants total approximately 700 lines of Alloy code. These models serve as the input to our tool along with a specification Φ and a set of predicates V[4].

In both variants there are 4 `Process`es: `Alice`, `Bob`, `Eve`, and a central `Server`. The attacker, `Eve`, can read all of the `Message`s exchanged between the `Process`es. The setup is similar to the running example that has been discussed throughout the paper. Both variants must satisfy the following specification.

Specification (Φ). We consider only one property across both variants of NSP: the secret `Key` K_{AB} shared between `Alice` and `Bob` is not leaked to `Eve`. We express this property as the following LTL formula:

$$\Phi = \mathbf{G}(K_{AB} \notin Eve.knows)$$

where $p.knows$ denotes the state variable of a protocol participant representing the set of `Keys` that the participant p has access to.

Symmetric. In the Symmetric variant, `Alice` notifies the `Server` that she would like to communicate with `Bob`. The `Server` then generates a communication key, `Key`$_{\mathsf{AB}}$, for `Alice` and `Bob` and sends it to `Alice`. This message is encrypted with `Bob`'s secret key. `Alice` forwards this message to `Bob` so that he will be able to decrypt the message with his secret key and learn the shared key. `Bob` then sends a random nonce to `Alice` that is encrypted with their shared key. `Alice` verifies that she knows the shared key by sending back `Bob`'s nonce decremented by 1 (Fig. 2).

Public-Key. In the Public-Key variant, `Alice` notifies the `Server` that she would like to communicate with `Bob`. The `Server` sends `Alice` a signed message with `Bob`'s public key. `Alice` sends `Bob` a message including a nonce that is encrypted with `Bob`'s public key. `Bob` receives this message and asks the `Server` for `Alice`'s public key. The `Server` sends `Bob` `Alice`'s public key. `Bob` now sends `Alice`'s nonce back to `Alice` along with a new nonce encrypted with `Alice`'s public key. `Alice` confirms that she has her private key by responding to `Bob` with his nonce encrypted with his public key.

Predicates. In the experiments described below, we used the following sets of predicates (V): Generic $= \{=, <\}$, consisting of only equality and one ordering predicate; $V_1 = $ Generic $\cup \{\mathsf{replay}\}$; and $V_2 = $ Generic $\cup \{\mathsf{manInTheMiddle}\}$. V_1

---

[4] The Alloy models and code for our tool can be found at https://github.com/cvick32/CounterexampleClassificiation.
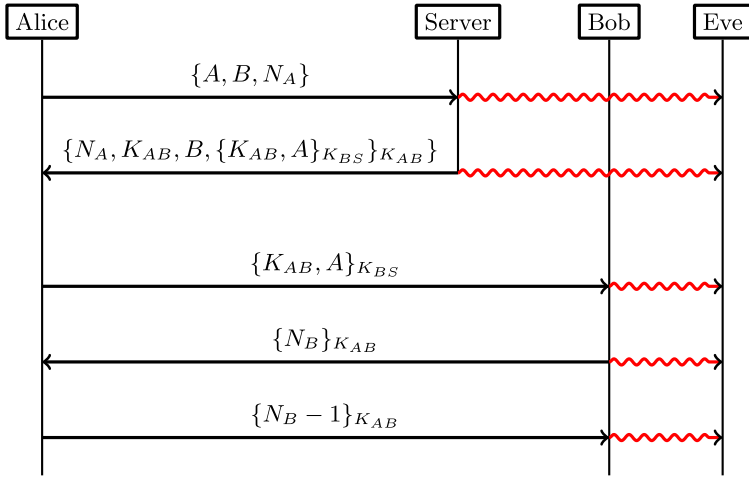
Fig. 2. A communication diagram of the Needham-Schroeder Symmetric protocol. A and B are identifiers for Alice and Bob respectively. There are three keys: K_{AB}, the shared key between Alice and Bob, K_{AS} and K_{BS} which are each Alice and Bob's server key. Alice and Bob also make use of a nonce, N_A and N_B respectively. Each arrow reprepresents a Message. $\{...\}_K$ denotes a Message encrypted by key K, and therefore requiring K to be read successfully. The snaking red lines represent Eve having access to all Messages that are sent over the network. (Color figure online)

and V_2 include all generic predicates plus some specialized predicates that characterize particular behavior in a model. The replay predicate, shown in Fig. 3, captures counterexamples where Eve sends the same message that was sent earlier by another process. The manInTheMiddle predicate captures counterexamples where Eve passes Alice and Bob's messages between them with no direct communication between Alice and Bob.

Predicates like replay and manInTheMiddle could be part of a library of predicates that any user could search and use. For example, replay can be used to check other communication protocols for replay attacks, provided that they follow a similar message-passing structure. Note that no information concerning the particularities of the Needham-Schroeder protocol is used in the definition of replay, meaning that this predicate can be used in a generic way. The same holds for manInTheMiddle.

Results. Our tool was able to produce classifications for both the Symmetric and Public-Key variants of NSP, as explained below. We were able to count up to 270,000 counterexamples (using the counterexample enumeration feature in Alloy) for both NSP variants until our program ran out of memory. The results are shown in Table 1.

Alloy employs bounded model checking for its verification engine; the *bound* column in Table 1 shows the upper bound used for the number of steps in traces explored by BMC. The V column shows the predicate set used in each experiment.

Input : A counterexample ρ and two time indexes $t1$ and $t2$
Output: A boolean
1 **pred** replay$[\rho, t1, t2]$:
2     $t1 < t2 \wedge$
3     $\rho.msg.sender.t1 \neq$ Eve $\wedge$
4     $\rho.msg.sender.t2 =$ Eve $\wedge$
5     $\rho.msg.nonce.t2 = \rho.msg.nonce.t1 \wedge$
6     $\rho.msg.process.t2 = \rho.msg.process.t1 \wedge$
7     $\rho.msg.key.t2 = \rho.msg.key.t1 \wedge$
8     $\rho.msg.encryption.t2 = \rho.msg.encryption.t1 \wedge$

Fig. 3. The replay predicate returns $\top$ if there are two positions $t1$ and $t2$ in ρ such that $t1$ occurs before $t2$ and the Message at $t1$ is the exact same as the Message at $t2$ except that Eve is now the sender.

The next column shows the number of classes generated and the last two columns show the execution time in seconds[5]. The execution time is split into the time our tool spent calling Alloy to find counterexamples and all other computations on the right. We found it instructive to show that the program was spending much of its time generating counterexamples in Alloy, while all other computations remained relatively constant for each respective experiment. Note that executions using V_2 take much longer than other executions. Most of this time is spent in generating the facts for manInTheMiddle as that particular predicate ranges over a number of time steps and all time steps in a counterexample must be checked. We also note that when using the Generic predicate set no redundant classes were found.

Table 1. Results on the Symmetric (left) and Public-Key (right) NSP variants. All times are recorded in seconds. All experiments were evaluated on a 2.5 GHz Quad-Core Intel i7 CPU with 16 GB of RAM.

bound	V	# classes	Alloy time	Total time
10	Generic	2	1.92	7.56
	V_1	3	4.29	10.23
25	Generic	2	9.37	16.24
	V_1	3	37.26	43.55
50	Generic	2	61.48	70.41
	V_1	3	220.49	226.37
75	Generic	2	254.38	267.96
	V_1	3	897.19	903.44
100	Generic	2	653.62	674.66
	V_1	3	1949.65	1955.133

bound	V	# classes	Alloy time	Total time
10	Generic	2	2.96	12.46
	V_2	3	6.01	100.61
25	Generic	2	12.96	24.04
	V_2	3	30.64	125.70
50	Generic	2	91.67	97.78
	V_2	3	157.62	251.89
75	Generic	2	321.95	349.21
	V_2	3	525.53	615.85
100	Generic	2	850.52	893.71
	V_2	3	1301.83	1396.92

Symmetric. This NSP variant is vulnerable to a replay attack. This attack has been addressed in implementations like Kerberos, although the attack was not found until 3 years after the initial publication of the protocol [5].

    Using the Generic predicate set, our tool generated 2 non-redundant classes. These classes characterize counterexamples where either Alice or Bob

---

[5] Times were measured using the Java built-in System.nanoTime().

unknowingly establishes communication with Eve, who then manages to extract the secret key from this interaction. For example, the trace constraint $TC_{Generic}$ shown below represents one of these two classes and characterizes counterexamples where Alice sends a message and at a later state, Eve manages to learn the secret key:

$$TC_{Generic}[\rho] \equiv \exists i_1, i_2 \in [0..len(\rho)] : \; \rho.msg.sender@i_1 = \texttt{Alice} \; \wedge$$
$$\rho.\texttt{Eve}.knows@i_2 = \{\texttt{Key}_{\texttt{AB}}\} \wedge i_1 < i_2$$

Although this constraint is a valid characterization of counterexamples (in that it is sufficient to guarantee a violation of Φ), it is rather an abstract one, in that it does not describe the intermediate steps that Eve carries out in order to extract the secret key.

To generate more specialized classes, the user can provide additional predicates beside the generic ones. Using V_1 as the predicate set, our tool generated 3 classes: the two classes previously found with Generic, plus a third class represented by the trace constraint TC_{Replay} shown below:

$$TC_{Replay}[\rho] \equiv \exists i_1, i_2 \in [0..len(\rho)] : \; \mathsf{replay}[\rho, i_1, i_2] \wedge i_1 < i_2 \; \wedge$$
$$\rho.msg.encryption@i_2 = \rho.msg.encryption@i_1 \; \wedge$$
$$\rho.msg.key@i_2 = \rho.msg.key@i_1$$

Our tool guarantees that we begin our classification with counterexamples that satisfy whichever predicate we choose, in this case replay. This is helpful as it constrains our classification to only those counterexamples which satisfy replay, allowing us to classify a subset of the total set of counterexamples. The constraint TC_{Replay} describes the type of violation where Eve carries out a replay attack, where she re-sends the message that was previously sent at step i_1 again at step i_2 with the identical message content. Note that although TC_{Replay} is a redundant class with respect to the other two classes generated using the generic predicates, it serves additional utility in that it provides more specific information about what Eve does in order to cause a security violation. The user of our tool (e.g., a protocol designer) could then use the information in these constraints to improve the protocol and prevent these types of violations.

Public-Key. This NSP variant is vulnerable to a man-in-the-middle attack [11]. Eve is able to forward messages between Alice and Bob and trick them into thinking they are communicating directly.

Similarly to the Symmetric variant, we were able to classify counterexamples that demonstrated the man-in-the-middle attack. The classes found in the Public-Key experiment reflected what we found in the Symmetric variant, i.e. 2 classes that show a general violating pattern with Generic and then 3 classes where 1 class demonstrates the known violation, using predicate set V_2. Our tool showed that the Public-Key variant is not vulnerable to replay attacks.

In summary, our classification method (1) significantly reduces the amount of information that the user needs to inspect to understand the different types

of violations, by collapsing the large number of counterexamples ($\geq$ 270,000 for the case study) into a small number of classes and (2) enables the user to inspect these different violating behaviors in a high-level representation (i.e., trace constraints) that can encode domain-specific information (e.g., replay attacks).

6   Related Work

It is well known that predicates can be used to abstract needless detail in certain problem domains [3,9]. This is the first time, to our knowledge, that predicates have been used for counterexample classification.

Our work can be considered a kind of automated debugging technique [16] in the context of model checking. There have been a number of prior works into locating the relevant parts of counterexample that explain or even *cause* a violation [1,2,7]. While our work does not deal with an explicit notion of causality, the generated trace constraints are sufficient to imply a violation of the property. The major difference between these works and ours is that they focus on *explaining* one or more given counterexamples, while our objective to *classify* the set of all counterexamples into distinct classes. Our work is also related and complementary to [10], which focuses on generating short counterexamples. We take a different approach by focusing on generating minimal trace constraints, each of which characterize a *set* of counterexamples.

The approach in [6] has the similar goal of generating a *diverse* set of counterexamples. This work relies on a notion of diversity that depends on general properties about the structure of the given state machine (e.g., counterexamples that have different initial distinct and final states). In comparison, our notion of diversity is *domain-specific*, in that it is capable of classifying traces based on domain-specific predicates that can be provided by the user. In this sense, these are two complementary approaches and could potentially be combined into a single model debugging tool.

7   Conclusion and Future Work

In this paper, we have proposed *counterexample classification* as a novel approach for debugging counterexamples generated by a model checker. The key idea behind our approach is to classify the set of all counterexamples to a given model and a property into *trace constraints*, each of which describes a particular type of violation. Our work leverages the notion of *predicates* to distinguish between different types of violations; we have also demonstrated how these predicates can capture violations that are common within a domain (e.g., attacks on security protocols) and can facilitate the reuse of domain knowledge for debugging.

For future work, we plan to explore methods based on machine learning (such as clustering (e.g., [14]) to automatically extract predicates from a given set of counterexample traces. Another interesting direction is to explore how our classification method could be used to improve counterexample-guided approaches to program synthesis (such as CEGIS [13]), by reducing the number of counterexamples that need to be explored by the synthesis engine.

References

1. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2003, pp. 97–105. Association for Computing Machinery, New York, January 2003

2. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 94–108. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_11

3. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1977, pp. 238–252. Association for Computing Machinery, New York, January 1977

4. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

5. Denning, D.E., Sacco, G.M.: Timestamps in key distribution protocols. Commun. ACM **24**(8), 533–536 (1981)

6. Dominguez, A., Day, A.: Generating multiple diverse counterexamples for an EFSM (2013)

7. Groce, A., Visser, W.: What Went Wrong: Explaining Counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–136. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44829-2_8

8. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. (TOSEM) **11**(2), 256–290. ACM, New York (2002)

9. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_15

10. Kashyap, S., Garg, V.K.: Producing short counterexamples using "Crucial Events". In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 491–503. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_47

11. Lowe, G.: An attack on the Needham-Schroeder public-key authentication protocol. Inf. Process. Lett. **56**(3), 131–133 (1995)

12. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (1978)

13. Solar-Lezama, A., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Combinatorial sketching for finite programs, p. 12 (2006)

14. Song, M., Günther, C.W., van der Aalst, W.M.P.: Trace clustering in process mining. In: Ardagna, D., Mecella, M., Yang, J. (eds.) BPM 2008. LNBIP, vol. 17, pp. 109–120. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00328-8_11

15. Torlak, E., Chang, F.S.-H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 326–341. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_23

16. Zeller, A.: The Debugging Book. CISPA Helmholtz Center for Information Security, 2021. Accessed 12 Mar 2021