Software and Systems Modeling
https://doi.org/10.1007/s10270-023-01118-0

SPECIAL SECTION PAPER

Counterexample classification

Cole Vick[1] · **Eunsuk Kang**[2] · **Stavros Tripakis**[3]

Received: 10 June 2022 / Revised: 26 June 2023 / Accepted: 28 June 2023

Abstract
In model checking, when a 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, (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 cover 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 and TCP protocols with promising results.

Keywords Model checking · Formal modelling · debugging

1 Introduction

In formal verification, *counterexamples* are an invaluable aid for debugging a 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, however, is 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 generates a finite number of *classes* that together cover the entire set of counterexamples, with each class being associated with a *constraint* that characterizes one particular type of viola-

Communicated by Antonio Cerone and Frank de Boer.

This work has been supported by the National Science Foundation under NSF SaTC award CNS-1801546.

✉ Cole Vick
cvick@cs.utexas.edu

Eunsuk Kang
eskang@cmu.edu

Stavros Tripakis
stavros@northeastern.edu

[1] Univeristy of Texas at Austin, Austin, TX, USA

[2] Carnegie Mellon University, Pittsburgh, PA, USA

[3] Northeastern University, Boston, MA, USA

Published online: 26 July 2023

tion. 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.

For instance, consider a security protocol involving a pair of agents that communicate over a channel, with an attacker that attempts to compromise the secrecy of exchanged information by carrying out various attacks. Although a model of the protocol may admit a large number of counterexample traces, each corresponding to a possible attack, suppose that each attack on this protocol can be classified as an instance of (1) a *man-in-the-middle* attack where the attacker places itself between the two agents or (2) a *replay* attack where the attacker resends a previously sent message. Given this model, our technique would automatically generate and present these two classes to the user together with representative counterexamples from each class.

A key idea behind our approach is the use of user-defined summary *predicates* for classifying counterexamples. In certain domains, the user may have a priori knowledge about common types of defects that can be encoded as generic constraints over system primitives. For example, there are well-understood categories of security attacks—e.g., man-in-the-middle and replay attacks—that can be expressed over concepts such as keys, messages, and agents. Our approach allows the user to use such constraints to control and fine-tune the result of classification. In addition, once defined, these predicates may be reused across multiple models within the same domain, as we demonstrate with the verification of security protocols in this paper.

We have built a prototype implementation of our classification technique on top of an existing formal modeling and verification tool, the Alloy Analyzer [12]. 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. In particular, our tool (1) leverages the capability of the Alloy Analyzer as a SAT-based bounded-model checker (BMC) to generate a counterexample, (2) uses a syntax-guided method to derive a class in the form of symbolic constraints, and (3) iteratively repeats steps (1) and (2) until it exhaustively explores all counterexamples of the original model.

As case studies, we have successfully applied our technique to two variants of the Needham–Schroeder protocol [19] and were able to classify hundreds of thousands of counterexamples into only a handful of classes that represent known attacks to the protocol. Additionally, we applied our technique to a model of the Transmission Control Protocol (TCP) and were able to classify known failures in 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 two case studies on two well-established distributed protocols, Needham–Schroeder and TCP (Sect. 5), that demonstrates the efficacy of our solution.

This paper extends the previous conference version of this work [27] with an additional case study on the widely used internet protocol TCP. Additionally, we include proofs for all theorems given in the original conference paper, give an algorithm for simplifying redundant solutions (Sect. 3.1.1), detail an important sub-procedure of our classification technique (Sect. 4.2), and present tool bug fixes and improved experimental results (Sect. 5).

1.1 Running example

To motivate our technique, we introduce the following example. `Alice` and `Bob` are sending `Messages` 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 $\texttt{Key}_{\texttt{AB}}$, Alice and Bob's shared key, to read `Encrypted` messages. `Eve` acquires $\texttt{Key}_{\texttt{AB}}$ 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 $Key = \{\emptyset, \texttt{Key}_{\texttt{AB}}\}$, and $EveSeenSecret$ of type Boolean, $\top$ for true and $\bot$ for false. $EveKey = \texttt{Key}_{\texttt{AB}}$ means that `Eve` has learned the key shared by `Alice` and `Bob`, while $EveKey = \emptyset$ means that `Eve` does not know the key. $EveSeenSecret = \top$ means that `Eve` has read a secret message. The initial state is $(\emptyset, \bot)$, meaning that `Eve` does not know the key and has not read any secret.

Transitions between states are labeled by `Messages`. 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]

[1] The traces in this section have labels, i.e., `Messages` 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.

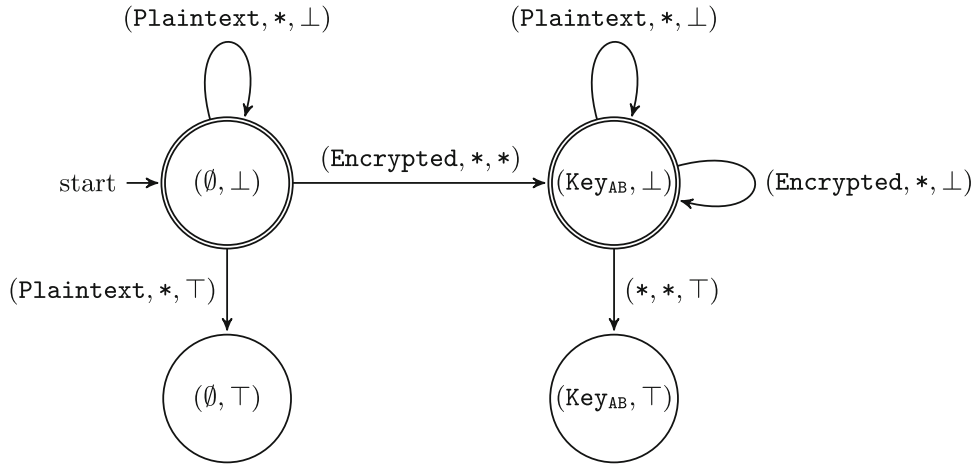

Fig. 1 Transition system of the running example

$*$ 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

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

which states that $EveSeenSecret = \bot$ 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 two top states satisfy the property, these are the *good* states, whereas the two states at the bottom of the figure do not, these are the *bad* or *error* states.

Note that this system has infinitely many *counterexample traces*, as self-loop transitions can be taken arbitrarily many times. Even when a system has a finite number of violating traces, presenting all of them to the user is not a good idea, as there are typically far too many to analyze. Keeping that in mind, some of the questions examined in this paper are the following: *How many of the violating traces should be presented to the user as counterexamples? Are some of these counterexamples* similar *in some sense? Can they be* classified *into some type of similarity* classes *so that only those classes are presented to the user?*

Take for instance the counterexample traces listed below:

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

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

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

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

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*. Note that this class contains not only the above four counterexamples, but an infinite number of distinct counterexamples where either `Alice` or `Bob` sends a `Plaintext Secret` message at any point in the trace. A potential succinct description of the class stated in words may be: `Eve` receives a `Plaintext Secret` message sent by `Alice` or `Bob`.

Now, consider the counterexample traces listed below:

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

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

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

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

$$\rho_2^5 = (\emptyset, \bot) \xrightarrow{(\texttt{Encrypted},\texttt{Alice},\bot)} (\texttt{Key}_{\texttt{AB}}, \bot) \xrightarrow{(\texttt{Encrypted},\texttt{Bob},\top)} (\texttt{Key}_{\texttt{AB}}, \top)$$

$$\rho_2^6 = (\emptyset, \bot) \xrightarrow{(\texttt{Encrypted},\texttt{Bob},\bot)} (\texttt{Key}_{\texttt{AB}}, \bot) \xrightarrow{(\texttt{Encrypted},\texttt{Bob},\top)} (\texttt{Key}_{\texttt{AB}}, \top)$$

$$\rho_2^7 = (\emptyset, \bot) \xrightarrow{(\texttt{Encrypted},\texttt{Alice},\top)} (\texttt{Key}_{\texttt{AB}}, \bot) \xrightarrow{(\texttt{Encrypted},\texttt{Bob},\top)} (\texttt{Key}_{\texttt{AB}}, \top)$$

$$\rho_2^8 = (\emptyset, \bot) \xrightarrow{(\texttt{Encrypted},\texttt{Bob},\top)} (\texttt{Key}_{\texttt{AB}}, \bot) \xrightarrow{(\texttt{Encrypted},\texttt{Bob},\top)} (\texttt{Key}_{\texttt{AB}}, \top)$$

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:

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

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: $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \cdots$.

We remark that the fact that the two classes above have a one-to-one correspondence with the two error states in the automaton of Fig. 1 is coincidental and not a feature of our technique. Later, we will present examples of classifications that break this correspondence both for this small example and our larger case study.

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 $\mathsf{Traces(STS)}$, 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 $\mathsf{Traces(STS)} \subseteq \Phi$. If $\mathsf{STS} \not\models \Phi$, then a *counterexample* is any trace $\rho \in \mathsf{Traces(STS)} \setminus \Phi$, 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 covering of P with (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 counterexample traces P and a classification C of P, a *canonical counterexample* ρ is any counterexample in P such that: $\forall c_1, c_2 \in C : (\rho \in c_1 \wedge \rho \in c_2) \rightarrow c_1 = c_2$. 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_5, 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:

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 $c \subseteq \bigcup_{i=1,...,n} c_i$.*

Proof ($\Leftarrow$) Suppose there exist distinct classes $c, c_1, ..., c_n \in C$ such that $c \subseteq \bigcup_{i=1,...,n} c_i$. We claim that c has no canonical counterexample. Indeed, take an arbitrary $\rho \in c$. Since $c \subseteq \bigcup_{i=1,...,n} c_i$, there must be some c_i such that $\rho \in c_i$. Moreover, c and c_i are distinct. Therefore, ρ cannot be canonical. Since ρ was chosen arbitrarily, there is no canonical counterexample in c, which means that C is redundant.

($\Rightarrow$) Suppose C is redundant. Then, there exists $c \in C$ such that c has no canonical counterexample. By definition, c is non-empty, so pick a $\rho \in c$. By assumption, ρ is not canonical. Therefore, there exists another class $c' \in C$, distinct from c, such that $\rho \in c'$. Let us denote c' by c_ρ, for any arbitrary ρ in c. Then, $c \subseteq \bigcup_{\rho \in c} c_\rho$. Moreover, the number of classes in C is finite, so even if c is an infinite set, the set of classes $\{c_\rho\}_{\rho \in c}$ is finite. Call that set $\{c_1, ..., c_n\}$. Then, $\bigcup_{\rho \in c} c_\rho = \bigcup_{i=1,...,n} c_i$, and thus, $c \subseteq \bigcup_{i=1,...,n} c_i$. □

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 $c_i \subseteq \bigcup_{j \neq i} c_j$.*

Proof ($\Leftarrow$) Follows directly from Lemma 1.

($\Rightarrow$) Suppose C is redundant. Then, by Lemma 1, there exist distinct classes $c_i, c_{k_1}, ...c_{k_m} \in C$ such that $c_i \subseteq \bigcup_{j=1,...,m} c_{k_j}$. But $\bigcup_{j=1,...,m} c_{k_j} \subseteq \bigcup_{j \neq i} c_j$, therefore, $c_i \subseteq \bigcup_{j \neq i} c_j$. □

3.1.1 Algorithm for non-redundant classification

Based on Lemma 2, we construct an algorithm to transform any classification into a non-redundant classification. Indeed, let C be a classification, where $C = \{c_1, ..., c_n\}$. First, we iterate over i and check whether there exists an i such that $c_i \subseteq \bigcup_{j \neq i} c_j$. If no such i exists, then, by Lemma 2, C is not redundant and we are done. If such an i does exist, then we remove c_i from C, to obtain the new classification $C_1 = C \backslash \{c_i\}$. Note that by removing c_i we do not run the risk of not covering the entire set of counterexamples P, since c_i is contained in the union of the remaining classes.

We continue in this way, removing any class that is covered by the union of all the other classes, until no such class exists, resulting in a non-redundant classification. The procedure is efficient because in the worst case, we perform no more than n checks of the form $c_i \subseteq \bigcup_{j \neq i} c_j$, where n is the number of classes in the original classification C.

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 constraints on those positions. For example, if $X = \{x, y\}$ is the set of state variables, then here are some examples of trace constraints:

- $TC_1[\rho] \equiv \exists i \in [0..len(\rho)] : x@i = y@i$: this trace constraint says that there is a position i in the trace such that the value of x at the position i is the same as the value of y at i.
- $TC_2[\rho] \equiv \exists i, j \in [0..len(\rho)] : i < j \wedge x@i > x@j$: 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 at i is greater than the value of x at 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 over 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 then be 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 ρ. 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 that satisfy 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 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., $\forall w \in W : c(w) \cap \Phi = \emptyset$.*

Proof Recall that a classification C of a set of $P = \text{Traces(STS)} \backslash \Phi$ must satisfy $\bigcup_{c \in C} c = P$ (Definition 7). Assume W is a solution to the CCP and there exists a $w \in W$ such that $c(w) \cap \Phi \neq \emptyset$. Thus, w accounts for some trace that is not in P. We have reached a contradiction because if this w were in W, then $\bigcup_{w \in W} c(w) \neq P$. □

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.*

Proof Recall that a trace is a finite sequence of states (Definition 3). Then, a trace $s_0, s_1, ..., s_k$ can be characterized by a conjunction of $k+1$ formulas, $\phi_0 \wedge \phi_1 \wedge \cdots \wedge \phi_k$, where each ϕ_i is itself a conjunction of atomic facts capturing state s_i. Specifically, let $X = \{x_1, ..., x_n\}$ be the set of state variables. Then, ϕ_i is of the form $x_1@i = v_1 \wedge x_2@i = v_2 \wedge \cdots \wedge x_n@i = v_n$, where v_j is the value of state variable x_j at state s_i. Notice that each ϕ_i has no position variables (Definition 10) because i is instantiated as a constant ranging from 0 to k. Indeed, in a fact such as $x_1@i = v_1$, i is the i-th position in the trace. It follows that $\phi_0 \wedge \phi_1 \wedge \cdots \wedge \phi_k$ is a trace constraint, without position variables. Therefore, a single trace can be characterized by a single trace constraint, and thus, such a trace constraint can also represent a class with a single trace in it. Therefore, if the set of counterexample traces is finite, we can have a classification represented by a finite number of trace constraints, one per counterexample trace. □

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.

Example 2 Consider an STS, shown in Fig. 2, 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.

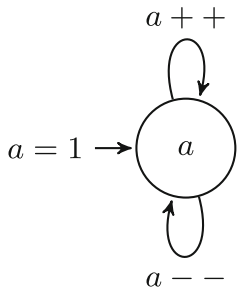

Fig. 2 Transition system for the increment-decrement example

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. Note 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:

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

□

We expect that our technique will be most useful in a bounded model checking environment where the number of counterexamples is finite and the equality operator is given. From Lemma 4, this guarantees that our technique terminates.

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 Sect. 3.3. We have just seen a possible solution $W_1 = \{TC_1, TC_2\}$. If we set V to $\{lessThanOne, greaterThanOne, \neq\}$, where $\neq$ is the not-equals predicate, the problem admits at least two solutions. Our predicate set allows the following trace constraint to be generated:

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

Note that TC_3 uses only the $\neq$ predicate to characterize the violating behavior. So, while W_1 is still a solution, a second solution $W_2 = \{TC_3\}$ is also possible.

Our approach admits both W_1 and W_2 as valid solutions to CCP, as it is difficult for an automated method to decide

which solution is "better." However, we show techniques in Sect. 4.3 that allow a user to fine-tune their solutions—e.g., by toggling redundancy checks and experimenting with different predicate sets.

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 an important sub-procedure used in classification (Sect. 4.2), describe optimizations to ensure the generation of a non-redundant classification with minimal classes (Sect. 4.3), and finally, present a solution to the Running example (Sect. 4.4).

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 counterexamples traces. This situation can be modelled by adding to V a trivial predicate that always returns $\top$, thus having a non-empty V. To guarantee termination, we assume that the set of counterexamples $P = \mathsf{Traces}(\mathsf{STS}) \backslash \Phi$ is finite. We also assume that $\mathsf{Traces}(\mathsf{STS}) \cap \Phi \neq \emptyset$.

```
   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 = ∅
3      while verify(STS ∧ block(W), Φ) == Violated do
4          ρ = counterexample(STS ∧ block(W), Φ)
5          Γ = facts(ρ, V)
6          if Γ = ∅ then
7              return "V cannot sufficiently characterize the
                violation in ρ"
8          w = traceConstraint(Γ, ρ)
9          if verify(STS ∧ w, ¬Φ) == Violated then
10             return "V cannot sufficiently characterize the
                violation in ρ"
11         w = minimizeTC(STS, w, Φ)
12         W = W ∪ w
13     W = removeRedundant(STS, W, Φ)
14     return W
```

Algorithm 1: The counterexample classification algorithm.

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

- verify$(\mathsf{STS} \wedge \varphi, \Phi)$: Returns OK if STS satisfies Φ under the additional constraint φ, i.e., if $\mathsf{Traces}(\mathsf{STS} \wedge \varphi) \subseteq \Phi$; else, returns Violated. The constraint φ is typically a trace constraint. We provide examples of φ later in this section.
- counterexample$(\mathsf{STS} \wedge \varphi, \Phi)$: If verify$(\mathsf{STS} \wedge \varphi, \Phi) ==$ 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 it does, obtains a counterexample that demonstrates how the violation occurs (line 4). The additional argument to the verifier, block(W), is used to prevent the verifier from re-generating a counterexample that is characterized by 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 positions 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 results in a violation of Φ. This guarantees that all traces that satisfy w are violations. If w does not guarantee a violation, 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)[2].

If w guarantees a violation, it is added to the set of classes that will eventually form a solution 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 counterexamples that have already been classified, classify passes block(W) as an additional constraint to verify, where:

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

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

[2] As an example where this happens, recall in the increment-decrement example from Sect. 3.3 how the single predicate $lessThanOne$ could not classify all violating behavior.

empty, as in the first iteration of the loop, $\mathsf{block}(W)$ returns $\top$.

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

Example 3 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 $W = \{TC_1, TC_2\}$. □

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.*

Proof We need to show (1) that each $w \in W$ is a trace constraint over the set of state variables X using predicates in V and (2) that $C(W)$ is a classification of the set of counterexamples P. (1) follows by construction from Algorithm 1. For (2), note that in order for $C(W)$ to be a valid classification of P, it has to cover all the traces in P. This follows from the fact that in order for W to be returned, Algorithm 1 needs to terminate, which means that the while loop on line 3 exits. This in turn implies that $\mathsf{verify}(\mathsf{STS} \wedge \mathsf{block}(W), \Phi)$ returns OK, which means $\mathsf{Traces}(\mathsf{STS} \wedge \mathsf{block}(W)) \subseteq \Phi$, which implies the result. □

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

Proof We need to show that no solution exists in each of the two cases when classify returns no solution.

Case 1: classify returns on line 7. This means that $\Gamma = \emptyset$, i.e., $\mathsf{facts}(\rho, V) = \emptyset$. From our assumption that there exists an accepting trace, ρ cannot be characterized by the empty trace constraint. Thus, ρ must be characterized by some non-empty trace constraint w. Such a w must contain at least one fact that ranges over X (the set of state variables of STS), uses predicates in V, and holds over ρ. But since $\mathsf{facts}(\rho, V) = \emptyset$, no such facts exist and therefore, w cannot exist.

Case 2: classify returns on line 10. This means that $\mathsf{verify}(\mathsf{STS} \wedge w, \neg\Phi)$ returns Violated, i.e., $\mathsf{Traces}(\mathsf{STS} \wedge w) \cap \Phi \neq \emptyset$. This in turn means that w does not guarantee a violation of Φ; that is, there are traces of STS that are characterized by w, and yet they satisfy the property Φ. Such traces are therefore not counterexamples. However, by Definitions 7 and 11, each generated trace constraint must characterize a subset of the set of counterexamples, i.e., $c(w) \subseteq P$. If w cannot guarantee a violation this means that $c(w) \nsubseteq P$, so w is not a valid trace constraint.

However, the fact that w is not valid does not immediately imply that there does not exist another trace constraint w' which is valid and characterizes ρ. Suppose such a w' exists; that is, suppose that (1) w' characterizes ρ and (2) $\mathsf{Traces}(\mathsf{STS} \wedge w') \cap \Phi = \emptyset$. For w' to characterize ρ, ρ must satisfy all the conjuncts of w'. Since every conjunct in w' was generated with V and ρ satisfies every conjunct, w should also contain each conjunct in w', by construction of facts and traceConstraint. Therefore, the set of conjuncts of w' is a subset of those of w, which means that w' is a weaker constraint than w. But this contradicts the facts that $\mathsf{Traces}(\mathsf{STS} \wedge w) \cap \Phi \neq \emptyset$ while $\mathsf{Traces}(\mathsf{STS} \wedge w') \cap \Phi = \emptyset$. Indeed, if w allows some traces of STS which satisfy Φ, and w' is weaker, then w' must also allow those traces, which means that $\mathsf{Traces}(\mathsf{STS} \wedge w') \cap \Phi$ cannot be empty. Thus, ρ cannot be characterized by any trace constraint and no solution can be found. □

4.2 Details on fact generation

The facts procedure is given in pseudo-code in Algorithm 2. For every predicate $v \in V$, and every state variable $x \in X$, where X is the set of state variables of the STS, we evaluate v with the given value of the state variable. Note that since state variables may change at each time step, the value at each time step must be checked.

Before a predicate is evaluated with a state variable, we check that the type of the state variable matches the type of the input to the predicate, using the typecheck procedure (line

```
Input  : a counterexample ρ and a set of predicates V
Output: set of true instantiations of each v ∈ V at each state in ρ
1 Func facts(ρ, V):
2     Γ = ∅
3     for v, x ∈ V, StateVariables(ρ) : typecheck(v, x) do
4         for i ∈ [0..len(ρ)] do
5             if eval(v(x@i)) then
6                 Γ = Γ ∪ v(x@i)
7     return Γ
```

Algorithm 2: The facts procedure.

3).[3] We finally evaluate the predicate with the state variable (line 5) and, if it returns $\top$, we add the expression to the set of facts and continue. Typically, Γ is specific to ρ, meaning that no other counterexample would satisfy the trace constraint generated from Γ, and as a result, Γ is not generalizable to other counterexample traces in the set of counterexamples, P. In Sect. 4.3.1, we will see how Γ is made to be generalizable.

Example 4 Consider the example given in Sect. 3.3, where we may either increment, decrement, or keep constant the single integer state variable, a. Take a sample counterexample $\rho = (1) \xrightarrow{--} (0)$ to the desired property $\mathbf{G}(a = 1)$. Since a typechecks on both predicates in V, 4 separate checks of $eval$ are performed. The checks $greaterThanOne[a@0]$, $greaterThanOne[a@1]$, and $lessThanOne[a@0]$ return $\bot$; $lessThanOne[a@1]$ returns $\top$, resulting in $\Gamma = \{lessThanOne[a@1]\}$. □

4.3 Optimizations

4.3.1 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 3. 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 3). 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 3, minimizeTC relies on the ability of certain verifiers, such as ones based on SAT [12] or SMT solvers [7], to produce a *minimal core* for the unsatisfiability of a formula [26]. 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 to imply a violation of Φ. However, if minimizeTC is invoked from line 9 in Algorithm 1, this side of the conditional branch is not reachable.

```
Input  : An STS, a trace constraint w, a specification Φ, and the
         set of facts Γ
Output: A minimized trace constraint
1 Func minimizeTC(STS, w, Φ):
2     if verify(STS ∧ w), ¬Φ) == OK then
3         γ = Γ ∩ minCore(STS, w, ¬Φ)
4         return traceConstraint(γ, ρ)
5     else
6         return "Γ does not sufficiently characterize the violation
           in ρ"
```

Algorithm 3: minimizeTC, which removes from trace constraint w all facts that are irrelevant to the violation depicted by ρ.

4.3.2 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

[3] The high-level algorithm shown is simplified as it only deals with unary predicates, but this technique can be extended to n-ary predicates. Our implementation is able to generate facts for predicates with an arbitrary number of arguments.

inspect.[4] 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 4: removeRedundant checks whether any $w \in W$ is redundant and if it is, removes it.

removeRedundant, shown in Algorithm 4, ensures that no trace constraint $w \in W$ is covered by any other trace constraints in W. This algorithm is an implementation in pseudocode of the process described in Sect. 3.1.1.

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 4). 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.

Recall the predicates $V = \{\neq, lessThanOne, greaterThanOne\}$ from Sect. 3.3. Suppose that classify finds two trace constraints in this order[5]:

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

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

4.4 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 $TC_{Plaintext}$ given the set of predicates $V = \{=, <\}$:

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

Equality, $=$, operates over Messages and Booleans while $<$ operates over position variables. Atomic position facts are generated just like atomic facts. Recall the following counterexample trace that is characterized by $TC_{Encrypted}$:

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

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 [12], 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. However, the tool relies on the SAT solver being able to compute minimal unsatisfiable cores, which are used for minimizing the trace constraints. Finally, our tool currently accepts only safety properties.

All experiments were evaluated on an AWS EC2 instance with 8GB of RAM and 2 vCPUs.

[4] Our tool allows users to toggle checking for redundancy as a user may want to inspect all generated classes, even if some may be redundant.

[5] Note that the newest trace constraint is never redundant because of the block procedure.

5.2 Case study: Needham–Schroeder

We applied our prototype to the well-known Needham–Schroeder protocol (NSP) [19], which has been known to be vulnerable to certain types of attacks [18]. 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 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.

5.2.1 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.[6]

In both variants, there are four `Processes`: `Alice`, `Bob`, `Eve`, and a central `Server`. The attacker, `Eve`, can read all of the `Messages` exchanged between the `Processes`. 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, illustrated in Fig. 3, `Alice` notifies the `Server` that she would like to communicate with `Bob`. The `Server` then generates a communication key, $\texttt{Key}_{\texttt{AB}}$, 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.

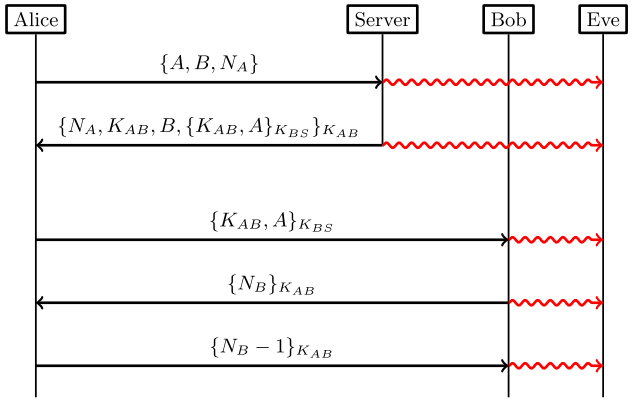

Fig. 3 A communication diagram of the Needham–Schroeder Symmetric protocol. A and B are identifiers for `Alice` and `Bob`, respectively. There are three keys: K_{AB}, the shared key between `Alice` and `Bob`, K_{AS} and K_{BS} which are each `Alice` and `Bob`'s server key. `Alice` and `Bob` also make use of a nonce, N_A and N_B, respectively. Each arrow represents 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

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 $\cup$ {replay}; and V_2 $=$ Generic $\cup$ {manInTheMiddle}. V_1 and V_2 include all generic predicates plus some specialized predicates that characterize more specific behavior. The replay predicate, shown in Fig. 4, captures counterexamples where `Eve` sends the same message that was sent earlier by another process. The manInTheMiddle predicate, shown in Fig. 5, 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 com-

[6] The Alloy models and code for our tool can be found at https://github.com/cvick32/CounterexampleClassificiation.

munication 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 or manInTheMiddle, meaning that either predicate can be used in a generic way.

5.2.2 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. The next column shows the number of classes generated, and the last two columns show the execution time in seconds.[7] The execution time is split between the time our tool spent calling Alloy on the left and all other computations, such as generating facts from a counterexample, on the right. The time spent calling Alloy is included in the total time.

Note that executions using V_1 and V_2 take longer than the executions using the Generic predicate set. Most of this time is spent in finding counterexamples that satisfy replay and manInTheMiddle, respectively. These executions require that the traces satisfy either replay or manInTheMiddle because we want to find violations that can be classified with these predicates. We also note that when using the Generic predicate set, no redundant classes were found.

We call attention to discrepancies between Table 1 and the same table presented in the original conference paper [27]. A bug was found that incorrectly assigned the bound. This led to our tool reporting a classification for a higher bound than was actually computed. We have fixed this bug and report the new results.

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 three years after the initial publication of the protocol [8].

Using the Generic predicate set, our tool generated 2 non-redundant classes. These classes characterize counterexamples where either `Alice` or `Bob` 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 three 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 is able to guarantee 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. TC_{Replay} characterizes all counterexamples where `Eve` re-sends a message that was previously sent at step i_1 again at step i_2—i.e., a replay attack. TC_{Replay} is a redundant class with respect to the other two classes generated using the Generic predicate set. It is useful as it provides more specific information about what `Eve` does in order to cause a 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 [18]. `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, three classes where one 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 tool (1) significantly reduces the amount of information that the user needs to inspect to under-

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

Fig. 4 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, thus modelling a replay attack

Input : A counterexample ρ and two time indexes $t1$ and $t2$
Output: A Boolean
pred replay$[\rho, t1, t2]$*:*
$t1 < t2 \wedge$
$\rho.msg.sender@t1 \neq \texttt{Eve} \wedge$
$\rho.msg.sender@t2 = \texttt{Eve} \wedge$
$\rho.msg.nonce@t2 = \rho.msg.nonce@t1 \wedge$
$\rho.msg.process@t2 = \rho.msg.process@t1 \wedge$
$\rho.msg.key@t2 = \rho.msg.key@t1 \wedge$
$\rho.msg.encryption@t2 = \rho.msg.encryption@t1$

Fig. 5 The manInTheMiddle predicate returns $\top$ if there are six positions in ρ such that $t1$ through $t6$ are in order and `Eve` intercepts and resends all messages sent between `Alice` and `Bob`

Input : A counterexample ρ and six time indexes $t1$ through $t6$
Output: A Boolean
pred manInTheMiddle$[\rho, t1, t2, t3, t4, t5, t6]$*:*
$t1 < t2 < t3 < t4 < t5 < t6 \wedge$
`EveSendAliceMessageToBob`$[t1, t2] \wedge$
`EveSendBobMessageToAlice`$[t3, t4] \wedge$
`EveSendAliceMessageToBob`$[t5, t6]$

Table 1 Results on the symmetric (left) and public-key (right) NSP variants

bound	V	# classes	Alloy time	Total time	bound	V	# classes	Alloy time	Total time
10	Generic	2	2.45	15.62	10	Generic	2	3.97	23.60
	V_1	3	4.76	18.46		V_2	3	5.79	26.80
15	Generic	2	7.05	47.39	15	Generic	2	9.90	69.94
	V_1	3	13.67	61.85		V_2	3	14.23	79.15
20	Generic	2	22.42	126.01	20	Generic	2	29.13	194.18
	V_1	3	40.28	168.41		V_2	3	36.15	224.12
25	Generic	2	53.22	289.23	25	Generic	2	72.01	437.21
	V_1	3	97.97	384.83		V_2	3	84.66	469.85

All times are recorded in seconds

stand 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.

5.3 Case study: TCP

TCP is a widely used Internet protocol. TCP's three main stages are connection establishment, data transfer, and connection teardown. Our model focuses on connection establishment and teardown and elides details and specifications surrounding data transfer. We adopt the same high-level TCP model described in [28] and shown in Fig. 6, which represents the state space for a single TCP agent.

As with Needham–Schroeder, we implemented our model in Alloy. Our Alloy model runs two TCP agents, A and B, side-by-side thus modelling two agents attempting to connect. These agents communicate over two channels, one is read by B and written to by A, `AtoB`, the other is read by A and written to by B, `BtoA`. There are four distinct *writable messages* that may be inserted into a channel: SYN, ACK, SYN_ACK, FIN. There is an additional message $INIT$ that is in the channel in the initial state and stays in the channel until a process inserts a writeable message. Finally, we model an `Attacker` as a process that is able to insert any writeable message in either channel at any time.

Specification (Φ). We consider the following safety property for our TCP model:

$$\Phi = \mathbf{G}(A.state = Closed \rightarrow B.state \neq Established)$$

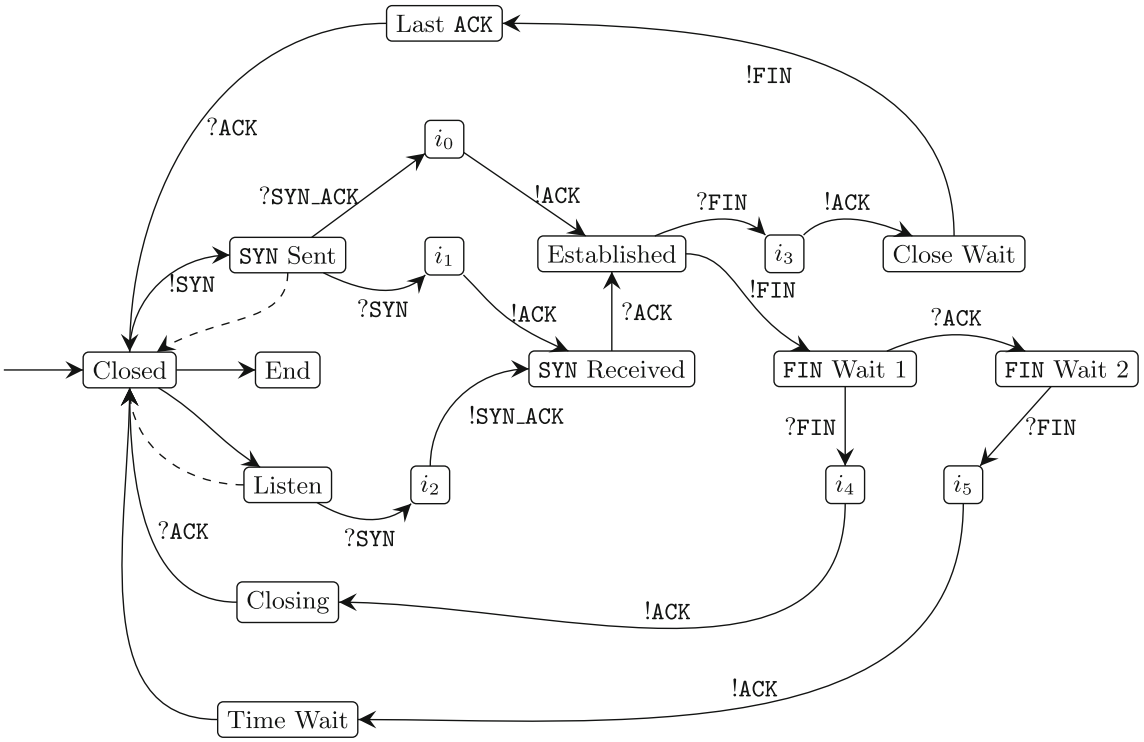

Fig. 6 The state machine for a single TCP agent. All the states except i_0, ..., i_5, and End are from the TCP RFC [20]. ! is read as "sends" and ? is read as "receives". The dashed transitions represent timeouts and can be taken wherever enabled. This figure is taken from [28]

So, for every point in our trace, it must be the case that if agent A is in `Closed`, then B must not be in `Established`. Note that our model satisfies this specification when we do not allow the `Attacker` to send messages.

Predicates. Similarly to the Needham–Schroeder case study, we evaluate our TCP model with two different sets of predicates. The `TCPGeneric` set is $\{=, <, AInState, BInState\}$ and V_3 is $\{=, <, AInState\}$.

Results. The results for our TCP experiments are shown in Table 2.

With V_3, we find only one class as follows:

$$TC_{V3}[\rho] \equiv \exists i_1 \in [0..len(\rho)] : \rho.A.state@i_1 = \texttt{ClosedState} \wedge \rho.B.state@i_1 = \texttt{EstablishedState}$$

Table 2 Results from the TCP experiment

bound	V	# classes	Alloy time	Total time
7	`TCPGeneric`	1	2.75	7.69
	V_3	1	0.96	4.67
8	`TCPGeneric`	1	8.11	26.43
	V_3	1	1.13	4.87
9	`TCPGeneric`	6*	N/A	N/A
	V_3	1	1.36	5.31
10	`TCPGeneric`	4*	N/A	N/A
	V_3	1	1.75	6.19

All times are recorded in seconds. N/A represents that the experiment ran out of memory. N^* means that we found N classes before running out of memory and the classes are not a complete solution to CCP

This single class is effectively a negation of the property, so intuitively, if a trace satisfies this property, then it will violate the specification. This example shows that depending on the selection of the predicates, it is possible for our tool to generate classes that do not have much explanatory power. To generate more "interesting" classes, the user may adjust the predicate set, as described next.

With the `TCPGeneric` predicate set, our tool is able to generate a complete classification for bound seven. For bounds of eight and above, our tool runs out of memory while attempting to generate a complete classification. However, when our tool is unable to find a complete classification, it still generates and outputs all of the classes it was able to find. In Table 2, using `TCPGeneric` results in finding a partial classification for runs with a bound greater than seven. Only 1 class is found using `TCPGeneric` for bounds seven and eight. This single class matches exactly the attack that was synthesized in [28], with the further guarantee that this is the only attack that the model is vulnerable to under this bound. At a high level, this class captures the `Attacker` acting as the active participant in an active-passive connection establishment. By sending a `SYN` message followed by

a SYN_ACK message, the Attacker can force B into the Established state. A must be in Closed when B reaches Established to guarantee a violation under the current Φ.

$$
\begin{aligned}
TC_{TCPGeneric7}[\rho] \equiv \exists i_1, i_2, i_3 \in [0..len(\rho)] : \\
\rho.B.read@i_1 = \texttt{SYN} \wedge \\
\rho.B.read@i_3 = \texttt{SYN\_ACK} \wedge \\
\rho.AtoB@i_1 = \texttt{SYN\_ACK} \wedge \\
\rho.AtoB@i_2 = \texttt{SYN\_ACK} \wedge \\
\rho.AtoB@i_3 = \texttt{SYN\_ACK} \wedge \\
\rho.B.read.i_3@sender = \texttt{Attacker} \wedge \\
\rho.A.state@i_1 = \texttt{ClosedState} \wedge \\
\rho.A.state@i_2 = \texttt{ClosedState} \wedge \\
\rho.A.state@i_3 = \texttt{ClosedState}
\end{aligned}
$$

$TC_{TCPGeneric9}$ is one of the six classes found when attempting to find a complete classification with a bound of nine. Since we were unable to find a complete classification, some of the classes that were found may be redundant. The underlying violation of $TC_{TCPGeneric9}$ is very similar to $TC_{TCPGeneric7}$. However, the more general $TC_{TCPGeneric7}$ is not sufficient to guarantee a violation under the increased bound because A and B have a larger execution space. With this larger execution space, it becomes harder to fully describe their behavior using shorter trace constraints like $TC_{TCPGeneric7}$.

$$
\begin{aligned}
TC_{TCPGeneric9}[\rho] \equiv \exists i_1, i_2, i_3, i_4, i_5, i_6 \in [0..len(\rho)] : \\
\rho.BtoA@i_1 = \texttt{INIT} \wedge \\
\rho.BtoA@i_3 = \texttt{INIT} \wedge \\
\rho.BtoA@i_4 = \texttt{INIT} \wedge \\
\rho.BtoA@i_5 = \texttt{FIN} \wedge \\
\rho.AtoB@i_4 = \texttt{SYN} \wedge \\
\rho.AtoB@i_5 = \texttt{SYN\_ACK} \wedge \\
\rho.AtoB@i_6 = \texttt{SYN\_ACK} \wedge \\
\rho.B.read@i_6 = \texttt{SYN\_ACK} \\
\rho.B.read@i_5 = \texttt{SYN} \\
\rho.A.state@i_1 = \texttt{ClosedState} \wedge \\
\rho.A.state@i_2 = \texttt{ClosedState} \wedge \\
\rho.A.state@i_3 = \texttt{ClosedState} \wedge \\
\rho.A.state@i_5 = \texttt{ClosedState} \wedge \\
\rho.B.state@i_1 = \texttt{ClosedState} \wedge \\
\rho.B.state@i_3 = \texttt{ClosedState} \wedge \\
\rho.B.state@i_4 = \texttt{ClosedState}
\end{aligned}
$$

For example, notice how much more specified the $TC_{TCPGeneric9}$ is as opposed to $TC_{TCPGeneric7}$. In particular, $TC_{TCPGeneric9}$ enforces which states A and B are in at particular time steps, while also enforcing the contents of both channels over five time steps. In contrast, $TC_{TCPGeneric7}$ does not constrain the state of B or channel BtoA at all.

Note that a user is able to parse these trace constraints directly and view counterexamples that satisfy particular trace constraints. For instance, they could execute a trace constraint and analyze the resulting counterexamples with the confidence that these counterexamples are characterized by the trace constraint. For instance, in Alloy a user could execute the above trace constraint with the command run $TC_{TCPGeneric9}$ for 9. The result of this command would be a sequence of visualized counterexamples that satisfy $TC_{TCPGeneric9}$.

We can also count the number of counterexamples that are satisfied by each counterexample class. We have not provided these results for all of the experiments as counting the number of counterexamples that a class covers often leads to timeouts, especially in the Needham–Schroeder case studies. However, there was an interesting case in our TCP case study where a single class covered well over 90% of the total counterexamples. In this case, we ran our TCP model with a bound of eight and found an incomplete covering of three classes before timing out. In total, there were 5572 counterexamples, 5468 of which were covered by the first class that was found. Counting how many counterexamples are covered by each class might be useful when deciding how to move forward when debugging a faulty model. In our tool, counting is done by enumerating the number of satisfiable instances Alloy is able to find. While this is not an exact count, it gives a rough estimate of the coverage of each counterexample class. Additionally, we are confident that advances in model counting may reduce the computational overhead for this kind of analysis in the future [10].

6 Related work

It is well known that predicates can be used to abstract needless detail in certain problem domains [6, 13]. 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 [29] 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, 3, 4, 11]. While our work is not based on 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 is to *classify* the set of all counterexamples into dis-

tinct classes. Our work is also related and complementary to [17], which focuses on generating short counterexamples. We take a different approach by generating minimal *constraints*, each of which characterize a *set* of counterexamples.

The approach in [9] 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.

Our approach is an application of the more general concept known as *abduction*—a type of reasoning method used to produce an explanation for a given observation about the world [14]. *Abductive logic programming* (ALP) is an extension of logic programming with the ability to perform abduction [15]. Although ALP has very different uses than model checking, in ALP systems such as HYPROLOG [5] and A-System [16], an explanation consists of a set of designated concepts called *abducibles*, which are similar to our notion of predicates.

Finally, the idea of using an unsatisfiable core for debugging models is not new and has been applied to debugging circuit designs [25], identifying over-constraints in declarative models [22], and minimizing a counterexample [21].

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 distributed 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., [24]—to automatically extract predicates from a given set of counterexample traces. We also plan to investigate further the impact of different predicate sets on the existence of solutions. Another interesting direction is to explore how our classification method could be used to improve counterexample-guided approaches to program synthesis (such as CEGIS [23]), by reducing the number of counterexamples that need to be explored by the synthesis engine. Additionally, the extension of our technique to the case of infinite counterexamples is still open.

A standard for sharing counterexamples and classifications is also left as future work. To accomplish this, the popular SMTLIB standard could be extended to provide a bridge between tools to share counterexamples and counterexample classes [2].

Data availability statement All of the material needed to reproduce the results from the paper are freely available in the Zenodo artifact published here: https://zenodo.org/record/7095162#.ZD7NS-zMJQI.

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'03, pp. 97–105. Association for Computing Machinery, New York, NY, USA, January 2003
2. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB). www.SMT-LIB.org (2016)
3. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: A Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, pp. 94–108. Springer: Berlin (2009)
4. Chechik, M., Gurfinkel, A.: A framework for counterexample generation and exploration. In: FASE, pp. 220–236 (2005)
5. Christiansen, H., Dahl, V.: Hyprolog: a new logic programming language with assumptions and abduction. In: ICLP, pp. 159–173 (2005)
6. 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 '77, pp. 238–252. Association for Computing Machinery, New York, NY, USA, January 1977
7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, pp. 337–340. Springer, Berlin (2008)
8. Denning, D.E., Sacco, G.M.: Timestamps in key distribution protocols. Commun. ACM **24**(8), 533–536 (1981)
9. Dominguez, A., Day, N., Cheriton: generating multiple diverse counterexamples for an EFSM (2013)
10. Gomes, C.P., Sabharwal, A., Selman, B.: Chapter 25. Model counting. In: Biere, A., Heule, M., Van Maaren, H., Walsh, T. (eds.) Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam (2021)
11. Groce, A., Visser, W.: What went wrong: explaining counterexamples. In: Ball, T., Rajamani, S.K. (eds.) Model Checking Software. Lecture Notes in Computer Science, pp. 121–136. Springer, Berlin (2003)
12. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. (TOSEM) **11**(2), 256–290 (2002)
13. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 447–491. Springer, Cham (2018)
14. Josephson, S.G., Josephson, J.R.: Abductive Inference: Computation, Philosophy, and Technology. Cambridge University Press, Cambridge (1994)

15. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming. J. Log. Comput. **2**(6), 719–770 (1992)
16. Kakas, A.C., Van Nuffelen, B., Denecker, M.: A-system: problem solving through abduction. In: IJCAI, pp. 591–596 (2001)
17. Kashyap, S., Garg, V.K.: Producing short counter examples using "Crucial Events". In: Gupta, A., Malik, S. (eds.) Computer Aided Verification. Lecture Notes in Computer Science, pp. 491–503. Springer, Berlin, Heidelberg (2008)
18. Lowe, G.: An attack on the Needham-Schroeder public-key authentication protocol. Inf. Process. Lett. **56**(3), 131–133 (1995)
19. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (1978)
20. Postel, J.: Transmission Control Protocol. RFC 793, September 1981. Available at: https://www.rfc-editor.org/info/rfc793
21. Shen, S., Qin, Y., Li, S.: Minimizing counterexample with unit core extraction and incremental sat. In: VMCAI, pp. 298–312 (2005)
22. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging over constrained declarative models using unsatisfiable cores. In: ASE, pp. 94–105 (2003)
23. Solar-Lezama, A., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS), p. 12 (2006)
24. Song, M., Günther, C.W., van der Aalst, W.M.P.: Trace clustering in process mining. In: Ardagna, D., Mecella, M., Yang, J. (eds.) Business Process Management Workshops. Lecture Notes in Business Information Processing, pp. 109–120. Springer, Berlin (2009)
25. Sülflow, A., Fey, G., Bloem, R., Drechsler, R.: Using unsatisfiable cores to debug multiple design errors. In: ACM Great Lakes Symposium on VLSI, pp. 77–82 (2008)
26. 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: Formal Methods. Lecture Notes in Computer Science, pp. 326–341. Springer, Berlin, Heidelberg (2008)
27. Vick, C., Kang, E., Tripakis, S.: Counterexample classification. In: Calinescu, R., Păsăreanu, C.S. (eds.) Software Engineering and Formal Methods, pp. 312–331. Springer, Cham (2021)
28. von Hippel, M., Vick, C., Tripakis, S., Nita-Rotaru, C.: Automated attacker synthesis for distributed protocols. In: Casimiro, A., Ortmeier, F., Bitsch, F., Ferreira, P. (eds.) Computer Safety, Reliability, and Security, pp. 133–149. Springer, Cham (2020)
29. Zeller, A.: The debugging book. CISPA Helmholtz Center for Information Security (2021). Retrieved 2021-03-12 18:02:07+01:00

Cole Vick is a PhD Student at The University of Texas at Austin and is advised by Ken McMillan. He received his Bachelor's degree from Northeastern University where he was an NSF REU recipient under the supervison of Stavros Tripakis. Between graduating from Northeastern and enrolling at UT Austin, Cole was a research assistant for Dr. Tripakis and Eunsuk Kang at Carnegie Mellon University.

Eunsuk Kang is an Assistant Professor in the Software and Societal Systems Department, School of Computer Science at Carnegie Mellon University. He received a Ph.D. in Electrical Engineering and Computer Science from MIT, and a Bachelor of Software Engineering from the University of Waterloo in Canada. His research interests include software engineering, formal methods, system safety, and security.

Stavros Tripakis is an Associate Professor of Computer Science at Northeastern University. He received his Ph.D. degree in Computer Science at the Verimag Laboratory, Joseph Fourier University, Grenoble, France, and has held positions at the University of California at Berkeley, at the French National Research Center CNRS, at Cadence Design Systems, and at Aalto University. His research interests are in the foundations of software and system design, formal verification, and cyber-physical systems. Dr. Tripakis was co-Chair of the 10th ACM & IEEE Conference on Embedded Software (EMSOFT 2010), and Secretary/Treasurer (2009-2011) and Vice-Chair (2011-2013) of ACM SIGBED. His H-index is 55.