A language for explaining counterexamples

Ezequiel José Veloso Ferreira Moreira
Universidade do Minho, Braga, Portugal
INESC TEC, Braga, Portugal

José Creissac Campos
Universidade do Minho, Braga, Portugal
INESC TEC, Braga, Portugal

Abstract

Model checkers can automatically verify a system's behavior against temporal logic properties. However, analyzing the counterexamples produced in case of failure is still a manual process that requires both technical and domain knowledge. However, this step is crucial to understand the flaws of the system being verified. This paper presents a language created to support the generation of natural language explanations of counterexamples produced by a model checker. The language supports querying the properties and counterexamples to generate the explanations. The paper explains the language components and how they can be used to produce explanations.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Model Checking, NuSMV, counterexample, natural language explanation

Digital Object Identifier 10.4230/OASIcs.SLATE.2024.11

Funding *Ezequiel José Veloso Ferreira Moreira*: acknowledges FCT (Fundação para a Ciência e Tecnologia) grant 2023.01639.BD, funded by the ESF (European Social Fund) and PQDI (*Programa Demografia, Qualificações e Inclusão*).

1 Introduction

It is often important to guarantee that a given system behaves correctly under all possible operating conditions. One way to verify if operating requirements are met is through Model Checking [6]. This requires a formal representation of the system (a specification) and expressing requirements as temporal logic formulas (the properties). A model checker can then automatically verify the specification's behavior against the properties. Should the verification fail, the model checker will attempt to produce counterexamples. That is, behaviours of the specification that do not satisfy the properties being verified. Understanding these counterexamples is fundamental to understanding the failures of the specification and, by extension, of the system being specified.

However, the interpretation of counterexamples can be complex and time consuming, representing a barrier to Model Checking adoption. With the aim of simplifying this interpretation process, we are developing an approach to generate natural language explanations for counterexamples. The approach makes use of property specification patterns [8, 10] to identify templates of natural language explanations for the counterexamples produced. The templates need to query both the property and the counterexample to determine relevant information to include in the explanation. To that effect, a pattern-matching language was defined. This paper describes this language and how it can be used to produce explanations.

The paper is structured as follows: Section 2 presents the state of the art, Section 3 discusses Model Checking, focusing on temporal logic and the counterexamples produced, Section 4 introduces the explanation methodology, Section 5 explains the language that was created to support it, Section 6 present an example of usage, and Section 7 presents conclusions and possible future developments.

2 State of the art

Several approaches have been proposed to support the process of understanding Model Checking results. Kaleeswaran et al. [12] provide an overview of the start of the art on the topic, identifying the need to support non-experts in understanding model checking results and natural language explanations as a promising direction for research.

Several approaches aim to produce explanations in natural language, which can more easily be understood even by non-experts in Model Checking. Van den Berg et al. [13], for example, propose a tool to interpret a counter-example and produce a description of the error that caused the safety violation. Their work is aimed a the railway domain.

ASSERT [7] is another tool that produces explanations. It uses ACL2s to analyze a series of requirements stored in an ontology. Should there exist a conflict between the requirements, the counterexample produced by ACL2s is used to produce a controlled natural language explanation that points to the requirements in conflict, the variables involved in said requirements, the type of error detected, and the state that led to the conflict.

Another example is AnaCon [1], which analyzes normative texts to determine conflicts within said texts. This is done first by writing these texts in a controlled natural language form, which is then converted into the formal language CL. This formal representation of the normative texts is then analyzed using the CLAN tool, which will produce a counterexample should a conflict exist. This counterexample is then converted into the same controlled natural language in which the normative text was first written, thus producing an explanation for the counterexample.

Yet another example of this is presented by Lu Feng et al. [9], which produces a series of structured language sentences to explain violations detected in a robot mission plan. This is done by first specifying the path taken by the robot as a Markov decision process (MDP) and analyzing this MDP formally using the PRISM model checker. Should a violation occur, then a counterexample is produced, which is itself an MDP. The counterexample is then explained by using controlled natural language sentences, recreating step-by-step the movement of the robot that leads to the violation.

3 Formal Methods and Model Checking

Formal Methods focus on the specification, verification, and implementation of computing systems through rigorous mathematical methods [14]. These methods are typically used within safety and mission-critical areas, such as railways, avionics, and finance [11].

Model Checking is a Formal Methods technique that focuses on the formal verification of a system's specification to determine if its behavior satisfies a series of properties [5]. These properties are defined in a given temporal logic (in this paper, CTL [4]), and then the specification is verified using a model checker (in this paper, NuSMV [3]). If this verification fails (i.e., the specification's behavior does not satisfy the property), the model checker will attempt to produce a counterexample. That is, a sequence of states (a path) in the system's specification that violates the property being verified.

3.1 CTL properties

Model checkers typically work with finite state representations of systems (e.g., Kripke structures). These consist of a finite set of states and transitions between these states, capturing the possible behaviors of the system. States are decorated with attributes that

help describe the system state at each moment. A path corresponds to a sequence of states (with their attributes' values) representing a possible behavior of the model.

CTL (Computational Tree Logic) is a branching-time temporal logic that is used to express and reason about the behavior of a system. Unlike linear time logic, which considers a single timeline, CTL allows multiple potential futures (paths) to be considered at any given state. For this purpose, the logic has both quantifiers over paths and temporal operators combined into pairs. First, a path quantifier specifies the scope of the paths in which the property must be verified. Then a temporal operator specifies in which states of any given path the property must be verified.

The available path quantifiers are the operators **A** (all paths) and **E** (exists a path). If operator **A** is used, the subsequent property must be verified in every future path in the model that starts in the current state. If operator **E** is used, then at least one path must exist, starting from the current state, that verifies the subsequent property.

Regarding temporal operators, CTL supports the following operators: **G** (globally), **F** (finally), **X** (next), and **U** (until). If the **G** operator is used, then the property must hold in every state of the path; if the **F** operator is used, then the property must eventually hold at some point in the path; if the **X** operator is used, then the property must hold in the next state in the path. The **U** operator expresses that one property must hold until another property becomes true.

Pairing the path and temporal operators allows the expression of complex properties over the system's behavior. For example, the property **AG** ϕ means that ϕ must hold in every state of every possible path in the future, while if **EF** ψ is used, then there must exist at least one path that in a future state verifies ψ. Similarly, **A**[ϕ **U** ψ] checks that ϕ hold until ψ hold in every path, while **E**[ϕ **U** ψ] requires the existence of at least one path that verifies ϕ until ψ. The properties ϕ and ψ above can themselves be CTL properties, or simple propositional logic expressions over the attributes of the state.

Simple propositional logic expressions are atomic formulas expressed in propositional logic with equality. Besides the usual propositional connectives (not – !, and – &, or – |, implies – $\rightarrow$, iff – $\leftrightarrow$, *xor* and *xnor*), and equality and inequality operators (=, ! =, >, <, >=, <=), the language supports also testing for set membership (*in* connective), Using these connectives, we can now write properties such as **AG**((*power* > 10) & (*weight* < 9)) that expresses that in every state of every path the attribute *power* should have a value above 10 and the attribute *weight* a value below 9, or the property **EG**((*power* = 2) $\rightarrow$ (*weight* = 8)) that expresses that there exists a path where it is always true that the attribute *power* having a value of 2 implies that the attribute *weight* has a value of 8.

3.1.1 Counterexamples

As stated above, when the specification exhibits a behavior that does not satisfy the CTL property, the model checker attempts to produce a counterexample. Useful counterexamples are typically generated for safety properties. That is, properties expressing that undesirable states will not be reached or undesirable behaviors are not possible in the system. In those cases, the counterexample consists of a path that shows a behavior of the system that violates the property being verified (the undesirable state being reached or the behavior happening). It is important to note that while the counterexample contains a finite number of states, it does not necessarily have a finite length in that a counterexample can end in a looping suffix.

In the case of liveness properties, that is, properties that express that desirable states of behaviors are possible, generating useful counterexamples is harder, as the problem is the system's lack of appropriate behavior.

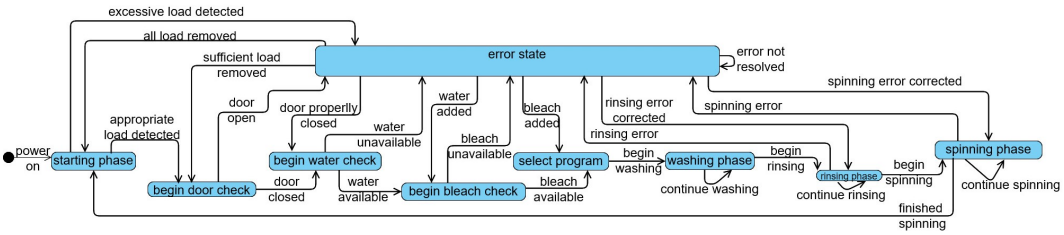

Figure 1 The state diagram that represents the ice cream machine's formal model.

Interpreting the produced counterexamples is a crucial aspect of performing model checking. While the verification itself is automated, the counterexample analysis must be done manually. The goal is to determine the cause of the problem. This entails, first, identifying, in the counterexample, what the problematic states or behaviors are, and second, why (or whether) these states or behaviors are problematic. The conclusion might be that the system's specification needs to be corrected, or it might be that the temporal logic property needs to be refined to better capture the intended requirement. Clearly, this analysis requires an understanding of both the system's specification and the system's domain.

3.2 An example

A simple example of a system is now presented to allow for greater clarity on the various concepts being discussed. The example used is a washing machine that can wash, rinse, and spin laundry, with various programs that determine how long each washing phase takes. To support this, the machine's specification defines a number of state attributes, including the current phase and its duration, the current state of the machine (starting a phase, ..., etc.), and an error code in case of malfunctioning.

We also define as one of its design requirements that *The machine must always either have its internal error code set to 0 or the machine's current state must be the error state.* Figure 1 presents the machine's control logic, focusing on the phase attribute.

To use a model checker (in this case, NuSMV) to verify this requirement, the first step is to specify the system formally. This was achieved using the IVY workbench tool [2], but discussing the model is outside this paper's scope. Herein, we are specifically interested in the counterexamples produced. It is enough to say that it represents the state machine in Figure 1.

With the model created, the requirement was converted into the CTL property:

$$\mathbf{AG}(error = 0 \mid state = errorState). \tag{1}$$

Attempting to prove this property fails, and the counterexample presented in Listing 1 is produced. The challenge now is to generate an explanation of the trace in natural language.

4 Explanation Methodology

The methodology aims to produce explanations for counterexamples generated by a model checker to simplify their interpretation. To achieve this, it uses three different sources of information (inputs) to generate the explanations: the counterexamples produced by the model checker (and corresponding properties), an explanation templates library, and domain information.

Listing 1 An example of a CTL property and resulting counterexample (in the required format to generate an explanation). Note that if an attribute has the same value between sequential states then that value is not shown, for size and readability reasons.

```
AG (error = 0 | state = errorState)         ---
--> State: 31.1 <-
  duration = none
  phase = standby
  door = open
  load = empy
  bleach = empy
  state = startingPhase
  program = 0
  error = 0
  water = empy
  action = nil
-> State: 31.2 <-
  load = overloaded
  state = errorState
  error = 4
  action = excessiveLoadDetected
-> State: 31.3 <-
  load = loaded
  state = beginDoorCheck
  action = sufficientLoadRemoved
```

The process of producing counterexamples starts with identifying a suitable explanation template consisting of partially filled-in sentences that will be completed with information from the various inputs to generate an explanation. In this section, we explain each of the inputs to the process and how they are used. A tool that implements the proposal is being developed and mentioned where deemed relevant.

4.1 Verification results input

The verification results input is used to provide the data related to the counterexample that needs to be explained.

Each item within this input consists of a pair (**property**,**counterexample**), where the **property** is the CTL property whose verification was attempted and failed, and the **counterexample** is the counterexample produced by the model checker for that **property**. An example of such an item can be found in Listing 1.

4.2 Explanation templates library

The explanation templates library provides the templates to produce the explanations for each property/counterexample pair. Listing 2 presents three examples of entries in this library.

The library contains a series of entries that consist of tuples (**pattern**, **conditionalTemplate1**, ..., **conditionalTemplateN**), where the **pattern** is used to dictate the types of **property** that can be explained using the entry, and each **conditionalTemplate** corresponds to a pair (**condition**, **template**). The **condition** is a logical expression over the counterexample that must hold for the corresponding **template** to be used for the

Listing 2 An example of three dicionary entries, with patterns that the property in Listing 1 matches

```
AG({E}{1,eq,x,y} | {E}{2,eq,x,y})    ----
Is should always be the case that either {DSE}{-,{E}{1}},
or {DSE}{-,{E}{2}}.
However, in state {S}{1,invert,sv} the two conditions do not hold,
with the verified values of {E}{1,-,x} and {E}{2,-,x} being
{S}{1,invert,t,{E}{1,-,x}} and
{S}{1,invert,t,{E}{2,-,x}}, respectively. ;;;;
AG({E}{1} | {E}{2})                  ----
It was expected that in every state of the machine
either {DSE}{-,{E}{1}} or {DSE}{-,{E}{2}}.\n
However on state {S}{1,invert,sv} neither
{DSE}{-,{E}{1}} nor {DSE}{-,{E}{2}}. ;;;;
AG({O}{1,or})                        ----
It was expected that in every state of the machine
{DSE}{-,{O}{1}} was always verified.\n
However at the end of the path {C}{2,3},
{DSE}{negateLeftSide,{O}{1}} ;;;;
```

explanation. It consists of a conjunction of sub-conditions expressed over the states of the counterexample.

A **template** will be used to explain the counterexample for a property if the **property** matches the **pattern** and the **condition** holds over the counterexample. As seen in Listing 2, a template consists of a mix of natural language and expressions that will query the property and counterexample for information. The language used to write these expressions (also used in the conditions above) will be discussed in the next section.

A special case occurs when there is only one **template** to be used if the **pattern** matches the property. For this specific case, the item becomes a simple pair (**pattern**,**template**) within the library. The three examples in Listing 2 fall in this category.

Should the data item not be matched by any **pattern**, or fail to meet all of the **conditions**, then the explanation can not be created, and the message given to the user will be *No matching pattern found for the specified property.*

4.3 Domain information input

This input is used to provide additional domain context that can be utilized during the explanation process. Examples of domain information entries are shown in Listing 3. They take the form of tuples (**domain pattern**, **contextExplanation1**, ..., **contextExplanationN**). The **domain pattern** identifies which expression in the specification can be explained using the particular entry. Each **contextExplanation** consists of a pair (**context**,**explanation**), where **context** is used to identify the context that the **domain pattern** is used in, and **explanation** is the explanation to be used in that context.

The notion of context enables control over how an expression captured by the domain pattern may be explained. Different contexts represent different ways to explain the same expression, depending on where they appear in the template. This enriches the explanations that can be produced, as it allows expressions to be described differently in different parts of the explanation. For example, differentiating between positive and negative contexts, or using long and short versions of the description to avoid repetition.

Listing 3 An example of four domain items

```
error = 0 ---- the current error code is 0, meaning no error
               is currently occurring ;;;;
error = {V}{x} ---- the current error code is {V}{x} ;;;;
state = errorState ---- the current state is the error state;;;;
error = 0 | state = errorState
\\\\ default ---- either the state is the error
        state,  or the current error code is 0.
\\\\ negateLeftSide ---- the current error code is not 0,
       and despite this the current state is the error state
\\\\ negateRightSide ---- the current state is the errorState,
       and despite this current error code is 0
;;;;
```

A special case for these items occurs when there is only one **context** with the value **default**. In this case, the item becomes a pair (**domain pattern**,**explanation**) within the domain input.

5 The language

As is clear from the discussion above, to instantiate the explanation templates, we need a means to query the CTL properties and the resulting counterexamples. To achieve this, we have defined a language that allows us, through pattern matching, to identify relevant elements in the verification results.

Several requirements for this language can be identified from the discussion. Regarding the writing of CTL patterns in the explanation templates library:

1. The language to be defined must support writing **patterns** to match properties written in the CTL language, as described in Section 3.1.
2. The tokens in the language, however, must only match propositional logic expressions. Temporal and path operators must be included explicitly in the patterns, not matched. This is because knowing the structure of the temporal operators in the property is relevant to deciding which type of explanation to provide. Allowing tokens in the language to match temporal sub-expressions would imply adding recursion to the natural language templates. While this would add expressive power, we feel the added complexity (both in terms of the writing of the explanations templates library and of its processing and use) is not justified.

Regarding the natural language templates:

3. The tokens in the language must allow for the explanation **template** to reference information in the property/counterexample pair whose property was matched by its associated **pattern**.
   a. The tokens must allow reference to the **property**, specifically to any part of it that has been matched using another token.
   b. The tokens must allow reference to any part of the **counterexample**, specifically any sequence of states, singular state, or specific attributes of any given state.
4. The language must allow for the **condition** in the explanation templates library to be expressed using the information in the **counterexample**. The goal here is to support tailoring the explanation to the counterexample produced when a property can fail in several ways.

5. The language must support the usage of domain information to replace information captured from the property or counterexample in an explanation **template**. This will allow the natural language explanation to be less technical and expressed in terms closer to the domain.
6. Similarly to Requirement 3, the language must allow the explanation **template** to reference the information matched by tokens in a domain item's **pattern**.

To fulfill these requirements, the language includes a series of tokens. All of the tokens consist of two parts, each encapsulated within **{}**: the type of the token, which dictates its main purpose, and the parameters, which modify the token's behavior. The types defined are **E** (Expression), **O** (Operation), **S** (State), **C** (Counterexample path), **DSE** (Domain Specific Explanation) and **V** (Value). The first two can be used in both the patterns and templates of the explanation patterns library. The following three are used in the templates only. The final one is used in domain items (both domain patterns and explanations).

Different token types have different required and optional parameters, but the parameters attached to each token type are the same independently of where the token is used. However, how the parameters modify the behavior of the token can change depending on where the token is being used. Parameters' values can be provided by naming the parameters explicitly (e.g., $\{O\}\{identifier : 1\}$), if names are not provided, the order of the values becomes relevant. For the sake of brevity, we will employ the unnamed variant (in this case $\{O\}\{1\}$).

5.1 Tokens in patterns

Tokens of type **E** and **O** can be used in the explanation templates library patterns to help define the type of property that will use a particular explanation template.

5.1.1 E tokens in patterns

The **E** token matches simple expressions only, with the option of further restricting what type of expression can be matched. This type of token has one required parameter, the **identifier**, as well as three optional parameters: **operation**, **nameV1**, and **nameV2**.

The **identifier** is a positive integer used to uniquely identify a token of this type, such that two **E** tokens with the same identifier must refer to the same simple expression. Thus, if the same **identifier** is used in two **E** tokens in a **pattern**, then the **property** being matched must have the same expression in both places, or else it will not match the **pattern**.

The **operation** parameter defines the type of simple expression the token will match. The values for this parameter are related to the available operators as follows: **eq** relates to $=$, **diff** relates to **!=**, **gt** relates to $>$, **lt** relates to $<$, **gte** relates to $>=$, **lte** relates to $<=$, **in** relates to **in** and **-** relates to any operation, serving as a wildcard for this parameter.

nameV1 and **nameV2** are both identifiers used to internally reference the values to the left and right of the operator, respectively, storing the values so that they may be explicitly used later in the explanation template. This allows for the explicit reference to each side of the expression and is necessary to fulfill Requirement 5.

Some examples of these tokens' use can be seen in Listing 2, such as with the second example's $\{E\}\{1\}$ or the first example's $\{E\}\{2, eq, x, y\}$. In the first case, the token is used to match any simple expression and will then be identified using the identifier 1. When the same token is used in the **template**, its value will be whatever was matched by the token in the CTL property. In the second case, the token will only match an equality expression, will use the identifier 2, and will store the left and right sides of the expression in the variables x and y respectively, such that their values can be accessed in the **template** explicitly.

With these options, the token can be used in the dictionary item's **pattern** to specify any type of simple expression that can be matched, partially fulfilling Requirement 1.

5.1.2 O tokens in patterns

The **O** token matches expression with propositional connectives only, again with the option of further restricting which connective can be matched. This token type has one required parameter, the **identifier**, and one optional parameter, the **operation**.

The **identifier** works as for the **E** token, being a positive integer that uniquely identifies the token. In the same manner, a **pattern** with two **O** tokens with the same identifier will only match **properties** that have the same exact expression in both places.

The meaning of **operation** differs, as it refers to the propositional connectives that the token can match. The values for this parameter are related to the connectives as follows: **not** relates to **!**, **and** relates to **&**, **or** relates to |, **imp** relates to **->**, **equi** relates to **<->**, **xor** relates to **xor**, **xnor** relates to **xnor** and **-** relates to any operation, serving as a wildcard for the parameter.

An example of this token's use can be seen in Listing 2, in the CTL property of the third example: $\{O\}\{1, or\}$. This token is used to match expressions with the connective |.

As illustrated, the token supports matching any propositional connective in the patterns of the explanation templates library's entries. This, in combination with the previous **E** token, allows for the fulfillment of Requirement 1. Since neither **O** type tokens nor **E** type tokens match temporal or path operators, Requirement 2 is also fulfilled.

5.2 Tokens in templates

The tokens that can be used in an explanation templates library entry's **template** are **E**, **O**, **S**, **C**, and **DSE**. Each of them represents a different part of the information captured in the verification results' pair to be added to the explanation.

5.2.1 E tokens in templates

Tokens of type **E** in a **template** serve to identify a simple expression that was matched in the **pattern** in order to use the captured information. The **identifier** parameter is used to determine which of the matched expressions the token refers to, such that an **E** token in the **template** will always refer to an **E** token in the **pattern** that has the same **identifier**.

Both the **nameV1** and **nameV2** parameters may be used in the **template** to refer to a specific side of the expression, provided that those same options exist in the corresponding **E** token that was used in the **pattern**. However, only a single one of these can be used per **E** token in the **template**, and if neither is specified, then the **E** token will refer to the complete simple expression. This occurs because the only context where it makes sense to use both sides is when talking about the whole expression.

The **operation** parameter is not relevant when the token is used in a **template**. The matching is done on the CTL property. Here, the goal is to refer to and use what has been matched. Hence, if used, the value of the **operaton** parameter will be ignored.

Some examples of these tokens can be seen in Listing 2, such as in the first example's template: $\{E\}\{1, -, x\}$ and $\{E\}\{1, -, y\}$. For both of these cases, the expression that is being referenced is the token $\{E\}\{1, eq, x, y\}$ in the **pattern**. In the first case, the expression will be replaced with the value stored in the x variable. In contrast, in the second case, it will be replaced with the value stored in the y variable, corresponding to the left and right sides of the matched expression, respectively.

With these options, the token can partially use any matched simple expression in the **template**, allowing it to partially fulfill Requirement 3a.

5.2.2 O tokens in templates

Tokens of type **O** in a **template** serve to identify a logical operation that was matched in the **pattern**. As before, the **identifier** parameter is used to determine which of the matched logical operations the token refers to, the same way as with the **E** token. Similarly, the **operation** parameter does nothing when used in the **template**, and its value will be ignored.

An example of these token's use can be seen in Listing 2, in the third example's template: $\{O\}\{1\}$. This token is used to refer to the **pattern** token $\{O\}\{1, or\}$, and will be replaced in the template with the value of the matched operation.

With these parameters, the token can use any logical operation in the **template**, as long as they can be matched in the CTL property. Combined with the previous **E** token, this allows for the complete fulfillment of Requirement 3a.

5.2.3 S tokens

Tokens of type **S** serve to query the **counterexample** for information regarding a specific state. This type of token has three required parameters, **index**, **order**, and **type**, as well as two optional parameters, **token** and **condition**.

The **type** parameter is used to distinguish what type of information the token will be replaced with in the template. If its value is **t**, then the replacement will use the state's attributes, while if the value is **sv**, then only the state's identifier will be utilized.

The **token** parameter can have any attribute name as its value and, when defined, will restrict the replacement made to only the value of the given attribute in the state. For this reason, this parameter only takes effect when the type parameter has value **t**.

The remaining parameters are used to identify which state the token refers to. In its simplest form, we can use the **index** and **order** parameters. **index** is a positive integer that acts as an index over the trace. **order** can be either **default** or **invert** and defines which end of the trace we want to count from. If the **order** parameter has value **default**, then the state the token is referencing will be the **index**$^{\text{th}}$ one of the counterexample. If the **order** parameter has value **invert**, then the state the token is referencing will be the **index**$^{\text{th}}$ last one of the counterexample.

Always counting states from the start or the end of the counterexample is too restrictive, so it is also possible to do it from a state that satisfies a given condition. To do this, we must use the **condition** parameter. This condition is defined as a conjunction of simple expressions and **E** tokens. When it is defined, then the first state of the counterexample where the expression becomes true is used as a reference, and the state the token is referencing be the **index**$^{\text{th}}$ one before or after it, depending on whether the **order** parameter is **default** or **invert**, respectively.

Several examples of these tokens can be seen in Listing 2. Two examples are the $\{S\}\{1, invert, sv\}$ and $\{S\}\{1, invert, t, \{E\}\{2, -, x\}\}$ expressions in the first item's template. In the first case, the token will be replaced using the state identifier for the first state that occurs counting from the end, i.e., with the identifier of the last state of the counterexample. In the second case, the token will be replaced with the value of the attribute represented by $\{E\}\{2, -, x\}$ (the left side of the expression $\{E\}\{2, eq, x, y\}$ that is captured in the **pattern**) on the last state on the counterexample.

With these options, the token can be used to reference states in the **counterexample** and obtain all its relevant data, fulfilling the part of Requirement 3b related to single states and specific attributes in any given state.

5.2.4 C tokens

Tokens of type **C** match sequences of states, i.e., paths, in the **counterexample**. This type of token has one required parameter, **start**, and two optional parameters, **end** and **token**.

The start **start** parameter defines the state where the path starts. It can have multiple different types of values, which affect the behavior of the token. If this parameter is a positive integer, then the start of the path will be that state in the **counterexample**. If this parameter is an **E** token, then the path will start at the first state of the **counterexample** where the simple expression the token represents first becomes true. If this parameter is the word **loop**, then the path will start at the first state of the loop within the **counterexample**. If this parameter is the word **full**, then the path will be the entire **counterexample**.

The **end** parameter defines the end of the path the token represents. It is always a positive integer, and if the **start** parameter is not **full**, then the path will end at the state indicated by this option. If the **start** parameter has value **full**, then the **end** parameter has no meaning.

The **token** parameter takes as value an attribute. If set, the parameter will change the replacement process for this token in the **template** so that only the value of the identified attribute will be present in each state of the path.

An example of this token's use can be seen in Listing 2, in the third example's template: $\{C\}\{2,3\}$. In this example, the token will be replaced by the path starting on the second state of the counterexample and ending on the third, with every state showing the values of all state attributes.

With these options, the token can be used to reference any sequence of states in the **counterexample** and obtain the relevant data, and in combination with the **S** token fulfilling Requirement 3b.

5.2.5 DSE tokens

Tokens of type **DSE** serve to replace parts of information in the **template** using their counterpart domain information. This type of token has two required parameters, **value** and **context**.

The **value** parameter can take as value an attribute, the value of an attribute, a simple expression, a propositional connective, an **E** token, or an **O** token. Its value will then be matched against a domain item's **domain pattern**, which will then be used to replace the token within the **template**.

The **context** parameter specifies the context that should be used when replacing the token in the template with a value from a context item of the domain information input. Its value is either an attribute or **-**. In the latter case, the value of the context becomes **default**.

Some examples of these tokens can be seen in the templates of Listing 2, such as with the first item's $\{DSE\}\{-,\{E\}\{1\}\}$ and the third item's $\{DSE\}\{negateLeftSide,\{O\}\{1\}\}$. For the first case, the token will be replaced by the domain item's information regarding the simple expression that is in the token $\{E\}\{1\}$ (i.e., the left side of the **or** expression in the CTL property). For the second case, the same will occur, using the **or** expression captured by $\{O\}\{1\}$, but instead of using the information in the **default** context, the explanation attached to the **negateLeftSide** context will be used instead.

With these options, the token can use domain information to replace any token being used in a template within the explanations templates library, thus fulfilling Requirement 5.

5.3 Tokens in the domain information

Only one type of token exists within the domain information, the **V** token. This token allows capturing values in a **domain pattern** and then using those values in the corresponding explanations. This type of token has one required parameter, **value**, and an optional parameter, **joinOperation**.

The **value** parameter is an identifier that is used to distinguish the data that was matched in the **domain pattern**, such that a **V** token with a different identifier must have matched a different value and vice-versa. This also means that if two **V** tokens have the same **value** in the **domain pattern**, then they must have matched the same value in multiple parts of said **domain pattern**.

The **joinOperation** parameter can either take the value **and** or the value **or** and is used to replace a list of values matched in the **domain pattern** with the enumeration of the values expressed as conjunction or disjunction, respectively. To be more precise, if the join operation is **and** and the list is [m1,m2,...,mn-1,mn], then the value will be replaced by "m1,m2,...,mn-1 and mn", with something similar occurring when using the **or** option.

An example of this token can be seen in Listing 3, in the second example: $\{V\}\{x\}$. In this example, the pattern will capture any simple expression whose left side has the value **error**, and the token will store the corresponding right side with the identifier x, allowing for the usage of this value in the corresponding **explanation** by using the same token.

The token is able to use information captured within the **domain pattern** in the paired **explanation**. This explanation will then be used to replace a **DSE** token in a **template**, thus fulfilling Requirement 6.

6 Using the language

With the language presented, we can finally showcase its use with the example presented in Section 3.2. A tool is being developed to support the process briefly described in Section 1. The tool is already able to read the verification results, an explanation templates library, and a domain information file and produce explanations for the counterexamples using the templates in the library.

To apply it to the example in Section 3.2, the property, as well as the counterexample that resulted from its verification attempt, were used as input to the tool (see Listing 1). To generate the explanations, an explanations templates library (see Listings 2 for an excerpt) and the domain input file in Listing 3 were used. With all the inputs available, the tool selected the first item of the dictionary that matched the **property**. In this case, the first item shown was selected.

The explanation was generated using this item by filling in the tokens within the **template**, according to the previously presented language. As such, the token $\{DSE\}\{-,\{E\}\{1\}\}$ was replaced by the domain information in the **default** context relating to the expression captured by the **pattern**'s token $\{E\}\{1, eq, x, y\}$, which corresponds to $error = 0$. In the domain input, the expression $error = 0$ only has a single context. Its value is *the current error code is 0, meaning no error is currently occurring*, and thus this phrase replaced the token $\{DSE\}\{-,\{E\}\{1\}\}$ in the final explanation. A similar process occurred for each of the DSE tokens, each being replaced with domain information relevant to each of the E tokens in the **pattern**.

Meanwhile, the **E** tokens are filled in with the information of the captured expression corresponding to the token's **identifier** parameter. In the case of $\{E\}\{1, -, x\}$ its value will be replaced with the value associated with the x variable in the **pattern**'s token $\{E\}\{1, eq, x, y\}$, which is the left side of the capture expression that has the value of *error*. A similar process occurs for the other **E** token, but referring to the **pattern**'s token $\{E\}\{2, eq, x, y\}$ instead.

The last type of tokens present in this example are the **S** tokens. For the first token, using the *sv* value for the type parameter leads to the token being replaced using the identifier of the corresponding state. This state is identified by the other two parameters and is the first to last state of the counterexample. Thus, the value that replaces this token is the identifier of the last state of the counterexample: 31.3.

For the other **S** tokens, the replacement uses the **type** t, meaning the replacement will be done with the values of attributes in the specified state (the last state). Additionally, the **token** parameter has an **E** token specified, corresponding to the left side of one of the captured expressions, an attribute. Thus, the value used in the replacement is the value of the attribute specified by the **E** token: *error* in the first case (with value 4) and *state* in the second (with value *beginDoorCheck*).

The explanation created for the example given using the shown inputs was finally:

> It should always be the case that either the current error code is 0, meaning no error is currently occurring, or the current state is the error state.
> However, in state 31.3 the two conditions do not hold, with the verified values of error and state being 4 and beginDoorCheck, respectively.

7 Conclusion and future work

A model checker can automatically verify a system's behavior against temporal logic properties. However, in case of failure, analyzing the results is still a manual process that requires both technical and domain knowledge. To aid this process, we have been exploring the generation of natural language explanations for model-checking counterexamples.

The approach we are developing makes extensive use of CTL property patterns and natural language templates to generate explanations for counterexamples. This paper describes the language used to query properties and counterexamples to instantiate the explanation templates. In particular, we have explained the tokens that allow for complex templates to be created and the reasoning behind their definition.

The paper reports on ongoing work and several venues for future work can be identified and are being pursued. One obvious area of future work is the continued improvement of the language. This should be guided by requirements arising from the need to create quality explanation templates. We plan to develop a series of case studies to analyze the expressiveness of the approach and, as part of this, to carry out user studies to assess the quality of the produced explanations.

In any case, several improvements have already been identified that will improve the expressive power of the language. We plan to expand the functionality of the **S** token when there is a condition so that it can search beyond only the first state where the condition is verified or allow for verification of conditions that occur in a state before/after the search point. Additionally, we want to expand the **S** token's **condition** so that it can also search for logical operations referenced by a **O** token.

Other enhancements include adding the ability to reference states obtained in a previous token to the current one, such that it is possible to check a condition only once and then refer to the state that matches that condition in other future tokens of the template; the

ability for the **token** parameter in both the **S** and **C** tokens to contain more than a single attribute; the ability for the **C** token to be able to have an **S** token for both the **start** and **end** options; or the possibility to automatically change a **DSE** token's context in a template based on a given condition.

Finally, we plan to explore how generative AI might be integrated into the approach and whether that can improve the quality of the generated explanations.

References

1 John J. Camilleri, Mohammad Reza Haghshenas, and Gerardo Schneider. A web-based tool for analysing normative documents in english. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 1865–1872, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167132.3167331.

2 J. C. Campos and M. D. Harrison. Interaction engineering using the ivy tool. In *ACM Symposium on Engineering Interactive Computing Systems (EICS 2009)*, pages 35–44, New York, NY, USA, 2009. ACM. doi:10.1145/1570433.1570442.

3 Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45657-0_29.

4 E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986. doi:10.1145/5397.5399.

5 Edmund M Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2008. doi:10.1007/978-3-540-69850-0_1.

6 Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

7 Andrew Crapo, Abha Moitra, Craig McMillan, and Daniel Russell. Requirements capture and analysis in assert(tm). In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pages 283–291, 2017. doi:10.1109/RE.2017.54.

8 M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. 21st International Conference on Software Engineering*, ICSE '99, pages 411–420. ACM, 1999. doi:10.1145/302405.302672.

9 Lu Feng, Mahsa Ghasemi, Kai-Wei Chang, and Ufuk Topcu. Counterexamples for robotic planning explained in structured language. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7292–7297. IEEE, 2018. doi:10.1109/ICRA.2018.8460945.

10 M.D. Harrison, P. Masci, and J.C. Campos. Verification templates for the analysis of user interface software design. *IEEE Transactions on Software Engineering*, 45(8):802–822, August 2019. doi:10.1109/TSE.2018.2804939.

11 Anne Elisabeth Haxthausen. *An Introduction to Formal Methods for the Development of Safety-critical Applications*. DTU Orbit, 2010.

12 Arut Prakash Kaleeswaran, Arne Nordmann, Thomas Vogel, and Lars Grunske. A systematic literature review on counterexample explanation. *Information and Software Technology*, 145:106800, 2022. doi:10.1016/j.infsof.2021.106800.

13 Lionel van den Berg, Paul Strooper, and Wendy Johnston. An automated approach for the interpretation of counter-examples. *Electronic Notes in Theoretical Computer Science*, 174(4):19–35, 2007. Proceedings of the Workshop on Verification and Debugging (V&D 2006). doi:10.1016/j.entcs.2006.12.027.

14 Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), October 2009. URL: https://doi.org/10.1145/1592434.1592436.