Fuzz Testing with Temporal Constraints

Klaus Havelund, Tracy Clark, and Vivek Reddy

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA

Abstract. Testing is in practice commonly performed by executing carefully planned scripts, which exercise particular planned scenarios. Fuzz testing can be applied as a complementary approach, to exercise a system outside the boundaries of the expected. However, fuzz testing can be too random. In this work, we explore an approach to fuzz testing using an expressive temporal logic named MaTL (Matching Temporal Logic) for expressing constraints on tests, restricting generated tests to only such that satisfy the provided constraints. MaTL is a linear temporal logic that supports future and past time operators and a match construct that selects argument values from events, similar to pattern matching in functional programming languages. Constraint solving is performed using the Z3 SMT solver.

1 Introduction

Our scenario is a software system, such as e.g. a spacecraft or rover, that is controlled by commands. A command is defined as a data record consisting of a name and a sequence of named arguments. Obviously, a wrong sequence of commands can have undesired consequences. Typically, such a software system, referred to as SUT (System Under Test), is tested by submitting carefully planned sequences of commands, referred to as tests, to it and verifying that observations of returned data satisfy expectations. This careful approach to test design is necessary. However, it may miss corner cases. It may be fruitful to produce more randomized tests, which humans have not designed the exact details of. Fuzz testing [32] can be used to exercise a system outside the boundaries of the expected in an attempt to break the system. In this paper, we explore, as a complementary approach to careful test design, how to fuzz test such systems by generating randomized tests (sequences of commands), constrained by temporal constraints, which can then be submitted to the SUT.

The given is a collection of predefined types of commands which can be submitted to the SUT, with their names and argument types. We refer to these as *command signatures*. A completely randomizing test generator will generate tests, each consisting of a completely randomized sequence commands, each with randomized arguments over the command signature. Although this may identify bugs in the SUT, it may be desirable to limit the randomness, and control it to avoid some scenarios, or dually, to explore some desired scenarios. To this end, we present both (i) a temporal logic, MaTL (Matching Temporal Logic), for constraining test generation, and (ii) an implementation in the form of the fuzz tool [9] that automatically generates such tests from a collection of MaTL

formulas. This allows a user to control the randomness by writing test specifications that restrict the generated tests to only those that satisfy a collection of formulas. Note that in this work, we do not explore how to evaluate the results of applying tests to the `SUT`. We assume that this is done in a separate activity, e.g. with runtime verification approaches [1, 8, 12, 17].

Specifically, MaTL is a linear temporal logic that supports future and past time operators, and a *match* construct, matching commands in a test, similar to pattern matching in functional programming languages. Given a collection of such temporal formulas, the Z3 SMT solver [31] is applied to generate tests that satisfy the formulas. However, although an SMT solver supports the generation of tests satisfying a formula, it may not generate sufficiently random tests satisfying the formula. For example, if a formula states that a test with 10 commands must be generated that contains at least one drive command, an SMT solver may generate a test with 10 drive commands. For this reason, we have to refine the tests generated by the SMT solver by randomizing as much as possible while still satisfying the constraints.

The paper is organized as follows. Section 2 introduces the MaTL temporal logic by defining its syntax and semantics. Section 3 provides a collection of example properties for testing a planetary rover. Section 4 explains how a first-order predicate logic SMT formula is generated from the collection of MaTL formulas. Section 5 explains how an SMT generated test is refined by randomizing its contents while still satisfying the SMT formula generated from the collection of MaTL formulas. Section 6 evaluates `fuzz` on the MaTL formulas presented in Section 3. Section 7 discusses related work. Finally, Section 8 concludes the paper.

2 The MaTL Temporal Logic

The MaTL temporal logic allows specification of future and past time properties, as well as matching on events (commands) and their data arguments. We shall use the terms *event* and *command* interchangeably. MaTL is interpreted on finite traces.

Traces. In order to get an intuition behind the logic to be presented, we need to define the models of the logic, namely traces, that the formulas in the logic denote. A *trace* is a finite sequence of events: $\langle e_1, e_2, ..., e_n \rangle$, where each event is a named record $id(id_1 = v_1, ..., id_n = v_n)$ with a name id, and $n \geq 0$ arguments named $id_1, ..., id_n$ with respective values $v_1, ..., v_n$. Later we shall see that commands are just a form of events.

Syntax. The core formulas of MaTL are defined by the following grammar:

$$
\begin{aligned}
\varphi &::= \mathtt{tt} \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle \mathtt{id(m)} \rangle \varphi \mid \mathtt{e} \diamond \mathtt{e} \mid \mathtt{e} \vdash \mathtt{r} \\
&\quad \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi \mid \ominus \varphi \mid \varphi \,\mathcal{S}\, \varphi \mid \Sigma^{+}_{[a,b]}\varphi \mid \Sigma^{-}_{[a,b]}\varphi \\
m &::= \epsilon \mid m\ m \mid id = p \\
p &::= x? \mid x \mid c \mid id.id \\
e &::= x \mid c \mid id.id \mid e \otimes e \\
\diamond &::= < \mid \leq \mid = \mid \neq \mid \geq \mid > \\
\otimes &::= + \mid - \mid \times \mid \div
\end{aligned}
$$

Here c can be an integer, a float, or a string, and a and b are non negative integers. In addition, parentheses are allowed with the obvious meaning. At a high level, this is effectively future and past-time LTL, extended with a few additional operators such as pattern matching over events (the most important one), and some more experimental ones such as matching strings against regular expressions, and counting how many times a formula is true.

The formulas $\mathtt{tt}$, $\neg\varphi$ and $\varphi \vee \varphi$ are the Boolean formulas for truth, negation and disjunction. We postpone the explanation of the match formula $\langle id(m) \rangle \varphi$ for a moment. The formula $e \diamond e$ compares the values of two arithmetic expressions using the classical relational operators. The formula $e \vdash r$ is true if the string value denoted by the expression e matches the regular expression r. Note that we have not specified the details of regular expressions. The formula $\bigcirc\varphi$ is true if φ is true in the next future state. The formula $\varphi_1 \ \mathcal{U}\ \varphi_2$ is true if φ_2 is true at some point in the future, and until then, not including, φ_1 is true. The formula $\ominus\varphi$ is true if φ is true in the previous state (the past dual of $\bigcirc$). The formula $\varphi_1 \ \mathcal{S}\ \varphi_2$ is true if φ_2 is true at some point in the past, and since then, not including, φ_1 is true (the past dual of $\mathcal{U}$). The formula $\Sigma^{+}_{[a,b]}\varphi$ is true if, over the entire future trace starting at the current position, the formula φ holds between a and b times ($0 \leq a \leq b$). The past-time dual $\Sigma^{-}_{[a,b]}\varphi$ is true if, over the entire past trace up to and ending at the current position, the formula φ holds between a and b times. Note that the interval $[a, b]$ refers to the number of trace positions where φ holds, not to a time window.

The match operator $\langle id(m) \rangle \varphi$ matches the current event being processed in the trace against $id(m)$, matching the event name against id and the event parameters against m as we shall see. In case $id =$ **any**, any command matches if the arguments match. The match must succeed and potentially yield new value bindings due to the parameters, after which the formula φ must be true at the same position in the trace, in the scope of these bindings. Recall that an event is a named record of the form: $id(id_1 = v_1, ..., id_n = v_n)$. The parameter match m is either the empty match ϵ (which means that there are no requirements on the parameters[1]), a match followed by a match $m\ m$, which means that both must match and their produced bindings are combined, or the matching $id_i = p_i$ of a specific parameter id_i against a pattern p_i. Patterns p can have one of four forms,

[1] Note that it does not mean that the event must have no parameters.

matching against a parameter value v. The pattern $x?$ causes the variable x to be bound to the value v of the parameter and is now in scope of the following formula φ. The pattern x is a variable already in scope, that must denote a value equal to the parameter v. The pattern c, a constant of integer, float, or string type, must be equal to v. The pattern $id_1.id_2$ denotes the enumerated value id_2 of the enumerated type $id_1 = \{..., id_2, ...\}$, and which must be equal to v. Enumerated types are used to define command argument types.

Derived Operators. We define the following derived operators.

$$\begin{array}{rll}
\mathtt{ff} = \neg\mathtt{tt} & \text{false} & (1)\\
\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2) & \text{and} & (2)\\
\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2 & \text{implication} & (3)\\
e_1 \diamond_1 e_2 \diamond_2 e_3 = e_1 \diamond_1 e_2 \wedge e_2 \diamond_2 e_3 & \text{ternary relation} & (4)\\
 & & \\
[id(m)]\varphi = \neg\langle id(m)\rangle\neg\varphi & \text{universal modality} & (5)\\
id(m) \Rightarrow \varphi = [id(m)]\varphi & \text{universal modality} & (6)\\
id(m) \bullet \varphi = \langle id(m)\rangle\varphi & \text{existential modality} & (7)\\
id(m) = \langle id(m)\rangle\mathtt{tt} & \text{event occurrence} & (8)\\
 & & \\
\Diamond\varphi = \mathtt{tt}\,\mathcal{U}\,\varphi & \text{eventually in the future} & (9)\\
\Box\varphi = \neg\Diamond\neg\varphi & \text{always in the future} & (10)\\
\Diamond\!\!\!-\,\varphi = \mathtt{tt}\,\mathcal{S}\,\varphi & \text{sometime in the past} & (11)\\
\Box\!\!\!-\varphi = \neg\,\Diamond\!\!\!-\,\neg\varphi & \text{always in the past} & (12)\\
\varphi_1\,\mathcal{U}_{\mathrm{w}}\,\varphi_2 = (\varphi_1\,\mathcal{U}\,\varphi_2) \vee \Box\varphi_1 & \text{weak until} & (13)\\
\varphi_1\,\mathcal{S}_{\mathrm{w}}\,\varphi_2 = (\varphi_1\,\mathcal{S}\,\varphi_2) \vee \Box\!\!\!-\varphi_1 & \text{weak since} & (14)\\
\bigcirc_{\mathrm{w}}\,\varphi = \bigcirc\varphi \vee \neg \bigcirc \mathtt{tt} & \text{weak next} & (15)\\
\ominus_{\mathrm{w}}\,\varphi = \ominus\varphi \vee \neg \ominus \mathtt{tt} & \text{weak previous} & (16)\\
 & & \\
\Sigma^{+}_{a}\varphi = \Sigma^{+}_{[a,a]}\varphi & \text{future exact count} & (17)\\
\Sigma^{-}_{a}\varphi = \Sigma^{-}_{[a,a]}\varphi & \text{past exact count} & (18)\\
\bigcirc_a\varphi = \bigcirc \bigcirc ... \bigcirc \varphi \quad \text{a times} & \text{a steps in future} & (19)\\
\ominus_a\varphi = \ominus \ominus ... \ominus \varphi \quad \text{a times} & \text{a steps in past} & (20)
\end{array}$$

Notes. Forms (1-4) are the standard Boolean logic forms. The universal modality (5) expresses that if the current event matches $id(m)$ then subsequently φ must hold. Note that this is different from the existential modality $\langle id(m)\rangle\varphi$ in the core syntax, which means that the current event must match $id(m)$ and thereafter φ must hold[2]. We have introduced alternative notations (6-7) for these modalities, reflecting more closely implication and sequential composition. (8) is yet a shorthand for just an event that occurs. The abbreviations (9-16) define the usual temporal logic operators for future and past time logic. The counting operators (17-20) should be obvious.

Example. Let us look at a few example properties. Consider the following trace consisting of three events:

[2] These opeartors are also known from modal logics, see e.g. [15, 19].

$$\langle open(file = \texttt{"log"}, mode = \texttt{"r"}), read(file = \texttt{"log"}), close(file = \texttt{"log"})\rangle$$

This trace satisfies the following three properties:

(1) $\Box(open(file = f?) \Rightarrow \Diamond close(file = f))$
(2) $\Box(read(file = f) \Rightarrow \neg close(file = f)\, \mathcal{S}\, open(file = f, mode = \texttt{"r"})$
(3) $\Sigma_1^+ open(mode = \texttt{"r"})$

Their meanings are as follows. (1) Whenever a file f is opened, it is eventually closed. (2) Whenever a file f is read from, in the past it has been opened in read mode, and not closed since. (3) There should be one open event in read mode.

Semantics. A MaTL formula denotes a set of finite traces as explained in the following.

Assignments. The bookkeeping of which variables are assigned to which event argument values is recorded in *assignments*, which map variables to values. Let X be a set of variables, and let V be a set of values that represent event arguments. An assignment $\gamma \in \Gamma = X \xrightarrow{m} V$ is a finite mapping from variables to values. We write $[x_1 \mapsto v_1, ..., x_n \mapsto v_n]$ to denote the assignment that maps each variable x_i to the value v_i. We denote by ϵ the empty assignment, also written $[]$. We denote by $\gamma[x \mapsto v]$ the assignment that differs from γ only by associating the value x to v. We denote by $\gamma_1 \dagger \gamma_2$ the assignment γ_1 overridden by the assignment γ_2.

Semantic Functions. We shall introduce some semantic functions used in defining the semantics of MaTL. First, we give some functions for which we do not give definitions since these are either trivial or out of scope of the paper. The function $\mathsf{Op} : \otimes \to V \times V \to V$ represents the semantics of arithmetic operators, mapping an operator $\otimes$ to a function that when applied to two numbers returns a number. The function $\mathsf{Rel} : \diamond \to V \times V \to \mathbb{B}$ represents the semantics of relational operators, mapping an operator $\diamond$ to a function that, when applied to two numbers, returns a Boolean. Finally, the function $\mathsf{Reg} : r \to V \to \mathbb{B}$ represents the semantics of regular expressions, mapping a regular expression to a function that when applied to a string returns a Boolean (match or not, regular expressions here do not bind values).

The function $\mathsf{Eval} : e \to \Gamma \to V$ represents the semantics of expressions, evaluating an expression e in an environment Γ, returning a value V. It is also fairly obvious and requires no additional comments, but we provide its definition since it involves variables.

$$\begin{aligned}
\mathsf{Eval}[\![x]\!]\,(\gamma) &= \gamma(x) \\
\mathsf{Eval}[\![c]\!]\,(\gamma) &= c \\
\mathsf{Eval}[\![id.id]\!]\,(\gamma) &= id.id \\
\mathsf{Eval}[\![e_1 \otimes e_2]\!]\,(\gamma) &= \mathsf{Op}[\![\otimes]\!]\,\left(\mathsf{Eval}[\![e_1]\!]\,(\gamma)\,,\, \mathsf{Eval}[\![e_2]\!]\,(\gamma)\right)
\end{aligned}$$

The function $\mathsf{Match} : m \to \Gamma \to C \to \Gamma_\perp$ is the core function. It gives semantics to the multi-argument pattern m occurring in a command pattern $id(m)$, given an environment Γ and a command C. The result is a new environment $\Gamma_\perp$, which is either a proper environment or $\perp$, which represents a failed match. The empty pattern ϵ matches all commands. Pattern composition $m_1\ m_2$ combines the results of each. Finally, the pattern $id = p$ matches if p matches the value of field id in the command χ.

$$
\begin{aligned}
\mathsf{Match}[\![\epsilon]\!]\,(\gamma)\,(\chi) &= [] \\
\mathsf{Match}[\![m_1\ m_2]\!]\,(\gamma)\,(\chi) &= \mathsf{Match}[\![m_1]\!]\,(\gamma)\,(\chi) \dagger \mathsf{Match}[\![m_2]\!]\,(\gamma)\,(\chi) \\
\mathsf{Match}[\![id = p]\!]\,(\gamma)\,(\chi) &= \mathsf{Pat}[\![p]\!]\,(\gamma)\,(\chi.id)
\end{aligned}
$$

The function $\mathsf{Pat} : p \to \Gamma \to V \to \Gamma_\perp$ is the other core function. It matches the argument pattern p against the argument value v. Note that a resulting empty environment [] denotes a match but without any new bindings.

$$
\begin{aligned}
\mathsf{Pat}[\![x?]\!]\,(\gamma)\,(v) &= [x \mapsto v] \\
\mathsf{Pat}[\![x]\!]\,(\gamma)\,(v) &= \textbf{if } v = \gamma(x) \textbf{ then } [] \textbf{ else } \perp \\
\mathsf{Pat}[\![c]\!]\,(\gamma)\,(v) &= \textbf{if } v = c \textbf{ then } [] \textbf{ else } \perp \\
\mathsf{Pat}[\![id_1.id_2]\!]\,(\gamma)\,(v) &= \textbf{if } v = id_1.id_2 \textbf{ then } [] \textbf{ else } \perp
\end{aligned}
$$

Trace Acceptance. Let σ be a finite trace of events of length $|\sigma|$ and i a natural number, where $0 \leq i < |\sigma|$. Then $(\gamma, \sigma, i) \models \varphi$ denotes that φ holds at position i of σ with the assignment γ. The formal semantics of MaTL is defined below. The most important formula is $\langle id(m)\rangle\varphi$, which matches the current event at position i against the command pattern $id(m)$, and requires φ to be satisfied in the same position in the scope of possible new variable bindings[3]. The semantics of the counting operators require that the cardinality of the set of trace positions for which the formula φ holds (in the future or past) be in the interval $a..b$.

- $(\gamma, \sigma, i) \models \mathtt{tt}$.
- $(\gamma, \sigma, i) \models \neg\varphi$ **iff not** $(\gamma, \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models \varphi_1 \vee \varphi_2$ **iff** $(\gamma, \sigma, i) \models \varphi_1$ **or** $(\gamma, \sigma, i) \models \varphi_2$.
- $(\gamma, \sigma, i) \models \langle id(m)\rangle\varphi$ **iff** $(name(\sigma(i)) = id$ **or** id = **any**$)$ **and**
  let $\gamma' = \mathsf{Match}[\![m]\!]\,(\gamma)\,\big(\sigma(i)\big)$ **in**
  $\gamma' \neq \perp$ **and** $(\gamma', \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models e_1\ \diamond\ e_2$ **iff** $\mathsf{Rel}[\![\diamond]\!]\,\big(\mathsf{Eval}[\![e_1]\!]\,(\gamma)\,,\,\mathsf{Eval}[\![e_2]\!]\,(\gamma)\big)$.
- $(\gamma, \sigma, i) \models e \vdash r$ **iff** $\mathsf{Reg}[\![r]\!]\,\big(\mathsf{Eval}[\![e]\!]\,(\gamma)\big)$.
- $(\gamma, \sigma, i) \models \bigcirc\varphi$ **iff** $i < |\sigma| - 1$ **and** $(\gamma, \sigma, i+1) \models \varphi$.
- $(\gamma, \sigma, i) \models \varphi_1\,\mathcal{U}\,\varphi_2$ **iff** $(\gamma, \sigma, j) \models \varphi_2$ **for some** $i \leq j < |\sigma|$
  and for all $i \leq k < j$ $(\gamma, \sigma, k) \models \varphi_1$.
- $(\gamma, \sigma, i) \models \ominus\varphi$ **iff** $i > 0$ **and** $(\gamma, \sigma, i-1) \models \varphi$.

[3] One could, as an alternative solution, argue that φ should hold in position $i+1$, which could then lead to a request for a dual past operator requiring φ should hold in position $i-1$.

- $(\gamma, \sigma, i) \models \varphi_1 \mathcal{S} \varphi_2$ **iff** $(\gamma, \sigma, j) \models \varphi_2$ **for some** $0 \leq j \leq i$ **and for all** $j < k \leq i$ $(\gamma, \sigma, k) \models \varphi_1$.
- $(\gamma, \sigma, i) \models \Sigma^{+}_{[a,b]}\varphi$ **iff let** $s = \{j \in [i..|\sigma| - 1] \mid (\gamma, \sigma, j) \models \varphi\}$ **in** $a \leq |s| \leq b$.
- $(\gamma, \sigma, i) \models \Sigma^{-}_{[a,b]}\varphi$ **iff let** $s = \{j \in [0..i] \mid (\gamma, \sigma, j) \models \varphi\}$ **in** $a \leq |s| \leq b$.

We say that a trace σ satisfies a formula φ, written $\sigma \models \varphi$, if and only if $(\epsilon, \sigma, 0) \models \varphi$.

3 Example

In this section, we provide an example, illustrating the input to the `fuzz` tool, and the results it produces. The input constraint language for the `fuzz` tool uses ASCII characters. The mapping from the symbols used in Section 2 to ASCII format is as follows. $\neg$: `not`, $\vee$: `or`, $\bigcirc$: `next`, $\mathcal{U}$: `until`, $\ominus$: `prev`, $\mathcal{S}$: `since`, $\Sigma^{+}_{[a,b]}$: `count (a,b)`, $\Sigma^{-}_{[a,b]}$: `countpast (a,b)`, $\wedge$: `and`, $\Diamond$: `eventually`, $\Box$: `always`, $\diamondsuit$: `once`, $\boxminus$: `sofar`, $\mathcal{U}_{\mathrm{w}}$: `wuntil`, $\mathcal{S}_{\mathrm{w}}$: `wsince`, $\bigcirc_{\mathrm{w}}$: `wnext`, $\ominus_{\mathrm{w}}$: `wprev`, Σ^{+}_{a} : `count a`, Σ^{-}_{a} : `countpast a`, $\bigcirc_{n}$: `next n`, $\ominus_{n}$: `prev n`. Note that all keywords also have ASCII symbol alternatives, although we do not use these here. For example `eventually` can be written as `<>`.

3.1 The Rover Command Example

We shall write constraints for generating tests for a planetary rover, which is controlled by commands. The rover can rotate, move (forward and backward), take pictures (of low, medium, or high quality), store them on a file system, send files to an orbiting satellite, collect samples (such as soil, rocks, etc.), and execute scripts. We operate with two enumerated types. The type `direction` represents the direction of driving, and the type `image_quality` represents the quality of pictures taken:

```
direction = { forward, backward }
image_quality = { low, medium, high }
```

The specific commands are as follows, written in an informal format, with arguments of types `uint` (unsigned integers), `float`, `string`, `direction` and `image_quality`. All commands have a number identifying the instance of that command, and a time stamp. Command numbers range from 0 to 1000 and time ranges from 0 to 10000. For some additional arguments, constraints are specified.

```
ROTATE(number:uint, time:uint, angle:float)
  where −180 ≤ angle ≤ 180
GOTO(number:uint, time:uint, x:float, y:float)
  where −10000 ≤ x,y ≤ 10000
MOVE(number:uint, time:uint, dir:direction, distance:float)
  where 1 ≤ distance ≤ 1000
PIC(number:uint, time:uint, quality:image_quality, images:uint)
  where 1 ≤ images ≤ 10
STORE(number:uint, time:uint, file:string, images:uint)
  where 1 ≤ images ≤ 10
SEND(number:uint, time:uint, file:string)
COLLECT(number:uint, time:uint, file:string, sample:string)
SCRIPT(number:uint, time:uint, script:string, file:string)
```

The command `ROTATE` rotates the rover an angle. The command `GOTO` moves to a given position (x, y) in the two-dimensional coordinate system that represents the planetary surface. The command `MOVE` moves the rover forward or backward a given distance provided in meters. The command `PIC` takes a given number of pictures of a given quality. The command `STORE` stores a given number of pictures on the camera in a file. The command `SEND` sends a file to an orbiting satellite (which can then send them to ground). The command `COLLECT` collects a given kind of sample and stores information about it in a file. The command `SCRIPT` executes a script given by name with a file as input.

3.2 Representation of Commands in XML

Command types (such as those above) are formally represented in XML files. The commands above are defined in an XML file, some of which is shown in Figure 1. The XML format was chosen because of its use in the context in which the tool was developed. We are looking into supporting shorter formats, including JSON and Yaml. The file shows the definition of the `direction` enumerated type and the `MOVE` command, and should be self-explanatory.

3.3 Running `fuzz`

The main function of the `fuzz` library is the `generate_tests` function, which has the following type:

```
Command = Dict[str, Union[int, float, str]]
Test = list[Command]

def generate_tests(spec: Optional[str] = None,
                   test_suite_size: Optional[int] = None,
                   test_size: Optional[int] = None) -> list[Test]:
```

The function takes as argument a specification `spec` of constraints, represented as a text string, how many tests to generate `test_suite_size`, and how many commands there shall be in each test `test_size` (all tests contain the same number of commands). The function returns a list of tests, each of which is a list of commands, each represented as a dictionary mapping a command field and parameter names to values. Arguments to the function are optional, with default values extracted from a configuration file, an example of which is shown here:

```
{
    "cmd_files": ["xml/rover_commands.xml"],
    "spec_file": "spec.txt",
    "test_suite_size": 10,
    "test_size": 10
}
```

We can run the test generator without any constraints as follows.

```
from fuzz import generate_tests

tests = generate_tests(spec='', test_suite_size=100, test_size=10)
for test in tests:
    for cmd in test:
        print(cmd)
```

```
<command_dictionary>
  <enum_definitions>
    <enum_table name="direction">
      <values>
        <enum numeric="0" symbol="forward"/>
        <enum numeric="1" symbol="backward"/>
      </values>
    </enum_table>
    ...
  </enum_definitions>

  <command_definitions>
    ...
    <fsw_command class="FSW" opcode="0x0003" stem="MOVE">
      <arguments>
        <unsigned_arg bit_length="32" name="number">
          <range_of_values>
            <include min="0" max="1000"/>
          </range_of_values>
          <description>Command number.</description>
        </unsigned_arg>
        <unsigned_arg bit_length="32" name="time" units="seconds">
          <range_of_values>
            <include min="0" max="10000"/>
          </range_of_values>
          <description>The dispatch time.</description>
        </unsigned_arg>
        <enum_arg bit_length="8" enum_name="direction" name="dir">
          <description>Direction to move.</description>
        </enum_arg>
        <float_arg bit_length="64" name="distance" units="meters">
          <range_of_values>
            <include min="1" max="1000"/>
          </range_of_values>
          <description>Distance to move.</description>
        </float_arg>
      </arguments>
    </fsw_command>
    ...
  </command_definitions>
</command_dictionary>
```

Fig. 1: XML representation of example command type.

This will in less than a second generate 100 completely random tests, one of which is the following (text strings are randomly generated and are here shortened to just `'...'`).

```
{'name': 'ROTATE', 'number': 51, 'time': 944, 'angle': -150.42}
{'name': 'SCRIPT', 'number': 75, 'time': 287, 'script': '...', 'file': '...'}
{'name': 'PIC', 'number': 64, 'time': 491, 'quality': 'low', 'images': 8}
{'name': 'SCRIPT', 'number': 48, 'time': 598, 'script': '...', 'file': '...'}
{'name': 'COLLECT', 'number': 39, 'time': 583, 'file': '...','sample': '...'}
{'name': 'COLLECT', 'number': 89, 'time': 78, 'file': '...', 'sample': '...'}
{'name': 'ROTATE', 'number': 87, 'time': 28, 'angle': 76.22}
{'name': 'GOTO', 'number': 90, 'time': 2, 'x': -8132.07, 'y': 4708.92}
{'name': 'PIC', 'number': 58, 'time': 531, 'quality': 'high', 'images': 6}
{'name': 'STORE', 'number': 74, 'time': 981, 'file': '...', 'images': 3}
```

We may conclude that such sequences are just too random, and that we might want to constrain their form. This is what is achieved by filling out the `spec` parameter with constraints.

3.4 Writing Constraints

We shall now write 10 constraints illustrating the constructs of the temporal logic. The first two constraints apply to all commands (**any**). The first constraint, named `p1`, specifies that for every command with a time argument t_1, if there is a next command (weak next), and it has a time value t_2, then it must hold that $t_2 \geq t_1 + 10$. In other words, time must progress with steps no less than 10. The second constraint, named `p2`, requires that command numbers increase by 1. Note how the values of arguments are bound using the `x?` notation. Note also that if a parameter is not mentioned, there are no constraints on it.

```
rule p1: # time increases
  always [any(time = t1?)] wnext [any(time = t2?)] t2 ≥ t1 + 10

rule p2: # command numbers are consecutive
  any(number = 1)
  and
  always [any(number = n1?)] wnext [any(number = n2?)] n2 = n1 + 1
```

The next two constraints, `p3` and `p4`, further constrain the arguments of the `ROTATE` and `MOVE` commands (compared to the constraints provided in the XML file in Figure 1).

```
rule p3: # rotation within range
  always [ROTATE(angle = a?)] −90 ≤ a ≤ 90

rule p4: # distance within range
  always [MOVE(distance = d?)] (d = 1 or d = 2 or d = 3)
```

The next two constraints are more complicated and illustrate the power of the match construct. The constraint `p5` specifies that if there is a `MOVE` command at a position in the test, going backward, and with a distance `d`, then eventually later in the test there must be a `ROTATE` command, rotating with an angle depending on the distance (simulating that the further backward the rover moves, the less it needs to rotate to get out of the situation it may be in). The next constraint, `p6`, states that every `GOTO` command must be followed only by `GOTO` commands that move "northeast" (increasing in both x and y).

```
rule p5: # move backward leads to rotation
  always [MOVE(dir = direction.backwards,distance = d?)]
    eventually <ROTATE(angle = a?)>
                    ((d ≤ 1 and a = 45) or (d > 1 and a = 20))

rule p6: # go northeast
  always [GOTO(x = x1?,y = y1?)]
    always [GOTO(x = x2?,y = y2?)]
      (x2 > x1 and y2 > y1)
```

Now were are adding a bit more complexity to the constraints. The constraint `p7` states that if `i` high quality images are taken, then eventually a subset `j` ($0 < j \leq i$) of those images must be stored in a file with a name matching the regular expression `\d\d\d\.img` (e.g. `134.img`), and after that, this file must be sent to the orbiting satellite.

```
rule p7: # image taking leads to storage and sending
  always [PIC(quality = image_quality.high, images = i?)]
    eventually <STORE(file = f?, images = j?)>
      (
```

```
      f ⊢ /\d\d\d\.img/
      and
      0 < j ≤ i
      and
      eventually SEND(file = f)
    )
```

The next constraint `p8` is an example of a past time property and shows how data values can also be related backward in time. The constraint states that if `j` images are stored in a file, then (from the previous step) this file must not have been stored before, and also there must have been taken a high quality picture in the past of a number of images `i` where $i \geq j$, and no file is stored since then except the current one.

```
rule p8: # image storing requires past image taking
  always [STORE(file = f?,images = j?)]
      prev (
        not once STORE(file = f)
        and
        (
          not STORE()
          since
          <PIC(images = i?, quality = image_quality.high)> i ≥ j
        )
      )
```

The next constraint `p9` illustrates how operations can be applied to data. It states that if a sample collection named `s` is collected and the analysis result is stored in a file `f`, then later, without any other collection in between, a script execution is commanded on that file `f`, where the script is named by concatenating `"run_"` with the sample name `s` and then `".py"`.

```
rule p9: # sample collection leads to script execution
  always [COLLECT(file = f?, sample = s?)]
   next (
     not COLLECT()
     until <SCRIPT(script = k?, file = f)> k = "run_" + s + ".py"
   )
```

All of our constraints so far have been on the form: "*if ... then ...*", which would be satisfied by a test even if none of the antecedents were true. In order to enforce some events to definitely happen, the last constraint `p10` states that we want to see a `MOVE`, a `PIC` with high image quality, and a `COLLECT` command.

```
rule p10: # required commands
  eventually MOVE() and
  eventually PIC(quality = image_quality.high) and
  eventually COLLECT()
```

3.5 Running `fuzz` with Constraints

If we now call `generate_tests` with this specification as follows:

```
spec = """
  rule p1: # time increases
    always [any(time = t1?)] wnext [any(time = t2?)] t2 ≥ t1 + 10
...
"""
tests = generate_tests(spec = spec, test_suite_size = 100, test_size = 10)
```

```
...
```

we will see a test like the following.

```
{'name': 'SCRIPT', 'number': 1, 'time': 1, 'script': '...', 'file': '...'}
{'name': 'PIC', 'number': 2, 'time': 11, 'quality': 'high', 'images': 10}
{'name': 'STORE', 'number': 3, 'time': 21, 'file': '...', 'images': 8}
{'name': 'SEND', 'number': 4, 'time': 31, 'file': '...'}
{'name': 'MOVE', 'number': 5, 'time': 41, 'dir': 'forward', 'distance': 1.0}
{'name': 'PIC', 'number': 6, 'time': 51, 'quality': 'high', 'images': 10}
{'name': 'STORE', 'number': 7, 'time': 61, 'file': '482.img', 'images': 7}
{'name': 'COLLECT', 'number': 8, 'time': 71, 'file': 'm', 'sample': 'm'}
{'name': 'SCRIPT', 'number': 9, 'time': 81, 'script': 'run_m.py', 'file':'m'}
{'name': 'SEND', 'number': 10, 'time': 91, 'file': '482.img'}
```

The reader is encouraged to verify that the test satisfies the above constraints. For example, wrt. rules `p1` and `p2` we observe that command numbers increase in steps of 1, and time progresses with steps no less than 10 – in fact, steps of 10, there is a limit to the randomness which will be discussed below. Another example is rule `p7` where we observe that commands number 2 and 6 request a high quality `PIC` to be taken, which is followed by a `STORE` command 7 in the file `482.img`, followed by a `SEND` command 10 of that file. Note that our constraints do not exclude two `PIC` commands that match one `SEND` command, and there are also other random `STORE` and `SEND` commands.

We also have to talk execution time. The first test is generated in less than a second, but subsequent tests take longer. In total, for 100 tests this sums up to 38 minutes for generating 100 tests, averaging 23 seconds per test. The reason for this low speed is the regular expression predicate `f ⊢ /\d\d\d\.img/` in rule `p7`. In general, string operations are very costly when using an SMT constraint solver such as Z3. If we remove this string predicate, `fuzz` generates the 100 tests in 68 seconds, averaging 0.7 seconds per test. Efficiency issues will be discussed in more detail in Section 6.

4 Generation of SMT Constraints

In this section, we shall try to give the reader an idea about the SMT formula generated from a user-provided specification in the MaTL temporal logic and how the SMT solver generates tests from it.

SMT solving. An SMT solver finds an assignment to free variables in a first-order predicate logic formula constraining the free variables. As a very simple example, consider the quantifier free formula $x > 10$ referring to the free variable x. If we ask an SMT solver to find a satisfying assignment to x it may come up with, e.g. $[x \mapsto 11]$. Let us look at a slightly more interesting example. Let $f : \mathbb{Z} \to \mathbb{Z}$ be a function, and consider the following constraint, stating that the function is injective:

$$\forall x, y \in \mathbb{Z},\ f(x) = f(y) \Rightarrow x = y$$

This problem can be encoded in SMT-LIB [26] (which has Lisp format) as follows.

```
(declare-fun f (Int) Int)

(assert
  (forall ((x Int) (y Int))
    (=> (= (f x) (f y)) (= x y))
  )
)

(check-sat)
(get-model)
```

The function f is referred to as an *uninterpreted function*. As a result, the SMT solver might return with the following assignment to f, which is basically the identity function $f(x) = x$ for all $x \in \mathbb{Z}$:

```
sat
(model
  (define-fun f ((x Int)) Int
    x)
)
```

Timelines and commands. We shall apply this very idea by considering a test, our free variable, as a time line function from natural numbers to commands, $timeline : \mathbb{Z} \to Command$, here programmed in Z3's Python API:

```
timeline: Function = Function('timeline', IntSort(), Command)
```

A test of size N (containing N commands) will be considered an assignment to the `timeline` variable, where we are only interested in the value returned by this function for the arguments $0..N-1$. So, e.g. $timeline(0)$ will denote the first command in the test, $timeline(1)$ the second command, and $timeline(N-1)$ the last command. The SMT constraint generated from the MaTL formulas will determine which assignments (tests) are assigned to the variable.

We first need to define the type `Command` of commands. This is done by parsing the XML file defining the commands shown in Figure 1. The definition of this datatype corresponding to our example is shown below in the SMT-LIB format, and consists of a list of constructors of the type, one for each command, and for each of these, a list of reverse selector functions, selecting the field values from a command of that type. As an example, let `R` be the command `(ROTATE 2 9873 45)` then `(ROTATE_angle R) = 45`. With each command C follows also a test predicate is_C, in this case `is_ROTATE(R) = true`.

```
(declare-datatypes ()
  (
    (Command
      (ROTATE (ROTATE_number Int)(ROTATE_time Int)(ROTATE_angle Real))
      (GOTO (GOTO_number Int)(GOTO_time Int)(GOTO_x Real)(GOTO_y Real))
      (MOVE (MOVE_number Int)(MOVE_time Int)
            (MOVE_dir direction)(MOVE_distance Real))
      (PIC (PIC_number Int)(PIC_time Int)
```

```
            (PIC_quality image_quality)(PIC_images Int))
        (STORE (STORE_number Int)(STORE_time Int)
               (STORE_file String)(STORE_images Int))
        (SEND (SEND_number Int)(SEND_time Int)(SEND_file String))
        (COLLECT (COLLECT_number Int)(COLLECT_time Int)
                 (COLLECT_file String)(COLLECT_sample String))
        (SCRIPT (SCRIPT_number Int)(SCRIPT_time Int)
                (SCRIPT_script String)(SCRIPT_file String))
    )
  )
)
```

Generated SMT formulas. We first schematically present the translation approach, ignoring in first instance the capturing of data values in events across the *timeline*. As we shall see, this boils down to simple indexing in the *timeline*. Assume two predicates p and q, and assume the temporal MaTL formula (always if p then eventually q):

$$\Box(p \rightarrow \Diamond q)$$

Note that this formula is equivalent to the MaTL formula $\Box([p]\Diamond q)$. If we consider the predicates to be predicates on commands, $p, q : Command \rightarrow \mathbb{B}$, we can interpret this temporal formula over our *timeline* function as follows, if we are interested in tests of length k.

$$\forall i \in \{0..k-1\},\ p(timeline(i)) \rightarrow \exists j \in \{i..k-1\}, q(timeline(j)))$$

We could use this formula as a guiding principle and translate our MaTL temporal logic formulas in this manner using universal and existential quantifiers. However, SMT solvers are known to be potentially inefficient on nested quantifiers as we have in this case. We therefore chose the alternative approach of unfolding the conditions as shown in the following formula for $k = 3$, which effectively states the same property ($\forall$ translates to $\wedge$ and $\exists$ translates to $\vee$ on a finite trace), recalling that $p \rightarrow q \equiv \neg p \vee q$.

$$\begin{aligned} & \neg p(timeline(0)) \vee q(timeline(0)) \vee q(timeline(1)) \vee q(timeline(2)) \\ \wedge\ & \neg p(timeline(1)) \vee q(timeline(1)) \vee q(timeline(2)) \\ \wedge\ & \neg p(timeline(2)) \vee q(timeline(2)) \end{aligned}$$

The same principle is applied for past-time temporal logic. To illustrate a more complicated constraint, we will show the SMT constraint generated from MaTL rule `p5` from Section 3.4, repeated here:

```
rule p5: # move backward leads to rotation
  always [MOVE(dir = direction.backwards,distance = d?)]
    eventually <ROTATE(angle = a?)>
                  ((d ≤ 1 and a = 45) or (d > 1 and a = 20))
```

To recall, it states that if we observe a `MOVE` command with direction `backward` and a distance `d`, then eventually we must observe a `ROTATE` command with an angle `a` that is a function of `d`.

We shall define the following shorthands. Let $\texttt{isM}(i) = \mathsf{is}(\mathsf{MOVE}, \mathsf{timeline}(i))$, $\texttt{isR}(i) = \mathsf{is}(\mathsf{ROTATE}, \mathsf{timeline}(i))$, $\texttt{Mdir}(i) = \mathsf{MOVE\_dir}(\mathsf{timeline}(i))$, $\texttt{Mdist}(i) = \mathsf{MOVE\_distance}(\mathsf{timeline}(i))$, and $\texttt{Rangle}(i) = \mathsf{ROTATE\_angle}(\mathsf{timeline}(i))$. `fuzz` generates a formula with the shape shown in Figure 2, for a test of size n. As in our small example above, we see n conjuncts. Each of these states that either the command at position i in *timeline* is not a MOVE command with direction backward, or (if it is), then one of the remaining disjuncts from position i to $n-1$ must be true, that is, that we observe a ROTATE command where the angle is correctly related to the MOVE distance observed at the time point i. Note how data capturing is represented by referring back to the position in the *timeline* (trace) where the data is collected.

$$\bigwedge_{i=0}^{n-1} \Big(\neg(\texttt{isM}(i) \wedge \texttt{Mdir}(i) = backward) \\ \vee \bigvee_{j=i}^{n-1} \Big(\texttt{isR}(j) \wedge ((\texttt{Mdist}(i) \leq 1 \wedge \texttt{Rangle}(j) = 45) \vee \ (\texttt{Mdist}(i) > 1 \wedge \texttt{Rangle}(j) = 20)) \Big) \Big)$$

Fig. 2: SMT constraint for rule p5.

Show me the code. The reader is referred to [9] for a full exposition of the `fuzz` implementation. Here we shall just extract a few definitions, which illustrate the main principles. MaTL rules are parsed using the Lark parsing toolkit [16] for Python. The parser generates from a specification an abstract syntax tree of Python class instances, representing language constructs used in the specification. As a representative example for a temporal operator, the class `LTLUntil` in Figure 3 represents the until formula $\varphi_1 \,\mathcal{U}\, \varphi_2$ (φ_1 until φ_2), requiring φ_2 to hold eventually and until then φ_1 must hold. This is represented as $\texttt{LTLUntil}(\varphi_1,\varphi_2)$.

The class defines two methods (among others not shown here). The method `to_smt` generates the SMT formula for this formula and exemplifies how these methods work. The method generates an SMT constraint according to the following pattern, here shown if we evaluate the formula at position 0:

$$\varphi_1 \,\mathcal{U}\, \varphi_2 = \varphi_2(0) \vee (\varphi_2(1) \wedge \varphi_1(0)) \vee (\varphi_2(2) \wedge \varphi_1(0) \wedge \varphi_1(1)) \vee ...$$

Since $\Diamond\varphi = \texttt{tt}\,\mathcal{U}\,\varphi$ one can easily see that the temporal formula $\Diamond\varphi$ will generate an SMT formula of the form $\varphi(0) \vee \varphi(1) \vee \varphi(2) \vee ... \vee \varphi(k-1)$. Note, however, that the implementation for efficiency reasons does not rewrite all formulas to the core logic.

While `to_smt` is used to generate traces that satisfy a formula, the other method, `evaluate`, works in the other direction and verifies that a trace satisfies formula $\varphi_1 \,\mathcal{U}\, \varphi_2$. For this formula, it checks that the current `index` is within the

trace and that φ_2 is true at `index`, or φ_1 is true, and then it recurses, checking that the formula is true for `index + 1`. This corresponds to the equation $\varphi_1 \mathcal{U} \varphi_2 = \varphi_2 \vee (\varphi_1 \wedge \bigcirc(\varphi_1 \mathcal{U} \varphi_2))$. The methods `evaluate` reflect the semantics in Section 2 without concern about efficiency. This turns out to be sufficient for our purposes since the scalability is determined by the `to_smt` methods, and the traces that can be generated using them can easily be verified with the simple `evaluate` methods in milliseconds. These methods are used by a function `verify_test(test: Test, spec: str) -> bool`: for testing all traces generated, providing an elegant automated testing technique. This function is also available to the user for testing user-generated tests.

```
@dataclass
class LTLUntil(LTLFormula):
  left: LTLFormula
  right: LTLFormula

  def to_smt(self, env: Environment, t: int, end_time: int) -> BoolRef:
    return Or(
      [
        And(
          self.right.to_smt(env, t_prime, end_time),
          And([self.left.to_smt(env, t_i, end_time)
                for t_i in range(t, t_prime)])
        )
        for t_prime in range(t, end_time)
      ])

  def evaluate(self, env: Environment, test: Test, index: int) -> bool:
    if within(index, test):
      return self.right.evaluate(env, test, index) or (
        self.left.evaluate(env, test, index)
          and
        self.evaluate(env, test, index + 1)
      )
    return False

  ...
```

Fig. 3: The LTLUntil class representing the formula $\varphi_1 \mathcal{U} \varphi_2$.

Another class is `LTLCommandMatch`, see Figure 4, which represents the pattern matching formula $\langle id(m) \rangle \varphi$. The `to_smt` method consists of five parts. First, it generates a formula `right_command` checking that the command is right (in the case of **any**, any command matches). Second, it generates a list `right_arguments` of formulas, checking that the actual arguments match the formal parameter specifications. This list is then composed in conjunction `event_constraint` with the formula that checks that it is the right command. Third, the environment `env_plus`, being a copy of the incoming environment, is built, mapping any binding variable names to the values on the time line at that time. Fourth, the subformula `subformula_constraint` is then evaluated in that new environment. Finally, the conjunction of `event_constraint` and `subformula_constraint` is returned. The method `evaluate` (not shown) operates in a very similar manner, except that it works with real values in the trace rather than with constraints.

```
@dataclass
class LTLCommandMatch(LTLFormula):
  command_name: str
  constraints: list[LTLConstraint]
  subformula: LTLFormula

  def to_smt(self, env: Environment, t: int, end_time: int) -> BoolRef:
    if self.command_name == 'any':
      right_command: BoolRef = True
    else:
      is_method: str = f'is_{self.command_name}'
      right_command: BoolRef = getattr(Command, is_method)(timeline(t))
    right_arguments: list[BoolRef] =
      [constraint.to_smt(env,t,end_time) for constraint in self.constraints]
    event_constraint = And([right_command] + right_arguments)
    env_plus = env.copy()
    bindings =
      [c for c in self.constraints if isinstance(c, LTLVariableBinding)]
    for binding in bindings:
      env_plus[binding.variable] =
        extract_field(binding.command_name, binding.field, timeline(t))
    subformula_constraint = self.subformula.to_smt(env_plus, t, end_time)
    return And(event_constraint, subformula_constraint)

  ...
```

Fig. 4: The LTLCommandMatch class representing the formula $\langle id(m) \rangle \varphi$.

The last class `LTLVariableConstraint` that we show, see Figure 5, represents argument matches of the form $command(..., field = x, ...)$ and is relatively self-explanatory.

```
@dataclass
class LTLVariableConstraint(LTLConstraint):
  variable: str

  def to_smt(self, env: Environment, t: int, end_time: int) -> BoolRef:
    actual_value =
      extract_field(self.command_name, self.field, timeline(t))
    return actual_value == env[self.variable]

  ...
```

Fig. 5: The LTLVariableConstraint class representing the argument match $command(..., field = x, ...)$.

5 Test Refinement

An SMT solver such as Z3 is not a great randomizer. It will find assignments that satisfy the constraints provided but will not necessarily attempt to randomize the assignments, either within an assignment or between assignments. Rather, it may try to minimize the job it has to do. As an example, suppose that we generate tests using only the two constraints `p1` and `p2` on Page 10, which require command numbers to increase by 1 and time to increase by at least 10. Our

`generate_tests` function will first ask Z3 to generate a test for these formulas, which (if we ask for 5 commands per test) may be (actual output):

```
{'name': 'SCRIPT', 'number': 1, 'time': 363, 'script': '', 'file': ''}
{'name': 'SCRIPT', 'number': 2, 'time': 373, 'script': '', 'file': ''}
{'name': 'SCRIPT', 'number': 3, 'time': 383, 'script': '', 'file': ''}
{'name': 'SCRIPT', 'number': 4, 'time': 393, 'script': '', 'file': ''}
{'name': 'SCRIPT', 'number': 5, 'time': 403, 'script': '', 'file': ''}
```

As we can see, it has generated a test consisting of five SCIPT commands, which satisfies the specification, but it is not very interesting. The `generate_tests` script therefore subsequently begins a refinement process, where it attempts to randomize each command, including its arguments, and if that fails, each argument of the original command. After each change, it calls `verify_test`, which calls the `evaluate` methods on the resulting test, and if it fails, that randomization is rejected. This procedure is described in Algorithm 1. After this process, it will generate a more randomized test, as e.g. the following (again, where random text strings have been replaced with `'...'`).

```
{'name': 'SEND', 'number': 1, 'time': 363, 'file': '...'}
{'name': 'SCRIPT', 'number': 2, 'time': 373, 'script': '...', 'file': '...'}
{'name': 'SEND', 'number': 3, 'time': 383, 'file': '...'}
{'name': 'COLLECT', 'number': 4, 'time': 393, 'file': '...', 'sample': '...'}
{'name': 'STORE', 'number': 5, 'time': 403, 'file': '...', 'images': 8}
```

The algorithm is a heuristic. One can approach this in different ways. A simpler version of Algorithm 1 was tried that consisted of not including lines 10-18 that randomize arguments. This solution was less effective. Another tried solution consisted, instead of lines 5-19, of invoking the SMT solver repeatedly with new random commands, and if it succeeded in generating a new test, the new command was kept. However, this turned out to be a very costly approach which does not scale well. Note that for each test, we call the SMT solver only once, with a new seed. We do not use a common approach of negating parts of the previous constraints used to obtain a new solution.

6 Evaluation

We evaluated test generation on (modifications of) the ten properties `p1`-`p10` introduced in Section 3.4. We performed two experiments. The first experiment used the scenario presented where some commands have string arguments, and where there are constraints on these (in particular properties `p7`, `p8`, `p9`). Strings turn out to be less efficient for SMT solving, and in particular regular expressions[4]. For this reason, we performed a second experiment, where we replaced string types with integer types and modified the formulas accordingly. This also includes enumerated types, which are represented as strings; these were mapped to numbers as well. This allows us to measure the price paid for operating with string constraints. The evaluation was carried out on an Apple MacBook Pro, with an M1 Max chip, and 64 GB of memory. The operating system was MacOS Sequoia, and applications were run in PyCharm 2023.1.2.

[4] In property `p7` we had to comment out the regular expression constraint `f ⊢ /ddd.img/` in order to get somewhat reasonable solving times.

Algorithm 1 Refine (further randomize) Test

```
Input: A MaTL specification and a test satisfying the specification
Output: A more randomized test still satisfying the specification
for each command c in the test do
    Replace c with a random command c'
    if the modified test satisfies the specification then
        Keep the replacement c'
    else
        Restore original command c
        # Randomize arguments:
        for each argument a of c do
            Replace a with a random argument a'
            # Test whether new argument works:
            if the modified test satisfies the specification then
                Keep the replacement a'
            else
                Restore original argument a
            end if
        end for
    end if
end for
return randomized test
```

For each property, we generated tests of increasing length (10 commands, 20 commands, etc.). We also solved for all properties together for different test lengths in the two experiments. We measured the SMT formula generation time, the SMT solving time, and the total execution time (approximately the sum of the formula generation and solving time). Results are shown as logarithmic scaled plots, with solid green curves (scenario without strings) and red dashed curves (scenario with strings). For each setting, we generated three tests per length in order to keep runtime manageable.

Figure 6 shows the combined execution time for solving all the ten properties. Figure 7 shows the execution time per property. Figure 8 shows the relationship between the formula generation time and the formula solving time for selected test sizes. Note that the overhead of test refinement (randomization) as described in Section 5 is negligible relative to formula generation and solving. All times reported in the figures are totals for generating three tests at each length. Formula generation is performed once per length, while SMT solving is repeated per test. Consequently, the average per test *SMT time* is $(T_{\text{total}} - T_{\text{formula-generation}})/3$.

The results clearly show the impact of string constraints on scalability. In the setting without strings, execution times grow moderately with test length, remaining within practical limits even for the combined case at lengths up to 100. In contrast, the with-strings setting shows a much steeper growth, with certain properties (notably `p7`-`p9`) exhibiting exponential blow-ups and the combined case becoming infeasible beyond length 50. The stacked bar plot further indicates that the dominant cost in the with-strings setting lies in SMT solving, while formula generation remains a relatively minor contributor. The results suggest that improving the handling of strings is essential for referring to strings in

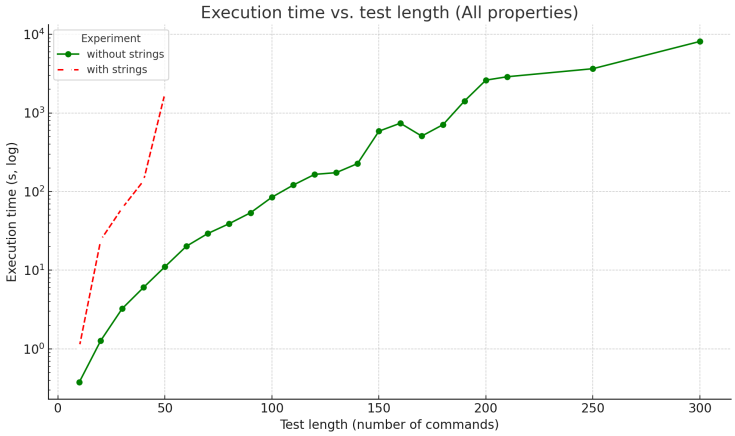

Fig. 6: Execution times (log scale) for all properties combined.

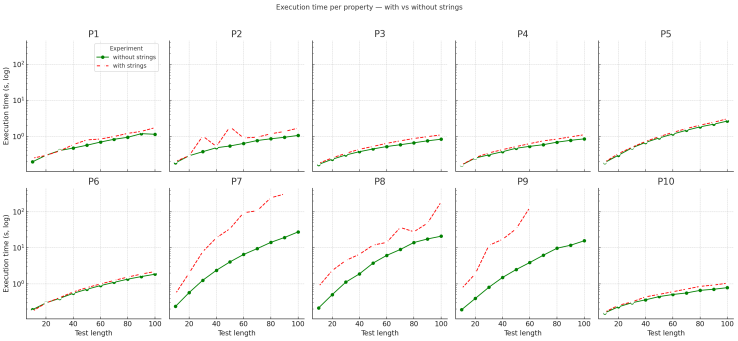

Fig. 7: Execution times (log scale) per property.

specifications. It is also clear that the approach has scalability issues on tests beyond 100 events. Note that scalability also depends on the number of commands involved. For test generation in practice, we anticipate that tests of around 50 commands are reasonable to expect, although the exact number will depend on the application scenario.

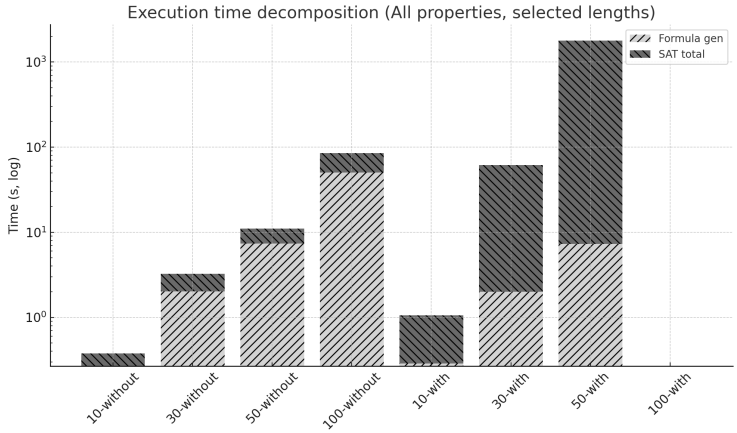

Fig. 8: Formula generation versus SMT solving time for selected test sizes (10, 30, 50, 100), without and with strings.

7 Related Work

The MaTL temporal logic is based on LTL [22], augmented with past-time operators, and a matching construct over events that carry data from potentially infinite domains, in the style of pattern matching in functional programming languages such as Scala [24]. The matching construct is related to the *freeze operator* in LTL logics and register automata over data words, see, e.g. [7], which supports storing data in registers in trace positions and comparing these with data in other trace positions with equality. MaTL goes beyond equality. Furthermore, register automata can store only as many data as the finite number of registered declared. The matching construct comes in two forms (*box modality* and *diamond modality*) corresponding to the modal operators of Hennessy–Milner logic [15,19]. Similar ideas appear in runtime verification logics [6,14,27], where the objective is to *monitor* traces rather than *generating* them. In particular, the Hawk logic [27] supports two modal operators similar to those provided by MaTL. MonPoly [2] and DejaVu [13] support (past) linear temporal logic with first-order quantification over data in events. The counting operators in MaTL correspond to similar operators in the temporal logic SALT [3].

The amount of work performed in the field of fuzz testing is overwhelming, and it will be impossible to provide a fair survey here. Generation of test cases from LTL is related to grammar-based fuzzing, which generates tests (strings) that satisfy a context-free grammar (rather than parsing such strings as grammars are normally used for). An excellent exposition on Python fuzz testing, including grammar-based fuzzing, is provided in [32]. Model-based testing uses higher-level models such as state machines. An example is [20], which translates SysML system models into bounded model checking problems, which are solved with SMT to produce tests. Our work can be seen as a form of model-based fuzzing, where the model is expressed in temporal logic. Constraint solving has long been employed for test generation. DART [11], CUTE [25], and KLEE [5] pioneered the concolic execution paradigm, where programs are run with con-

crete inputs while collecting symbolic path constraints, which are systematically negated and solved with an SMT solver to explore new paths beyond random fuzzing.

Generation of tests from an LTL formula is specifically closely related to checking satisfiability. Most work in this direction concerns future time temporal logics over atomic propositions (no data). Intuitively, if an LTL formula φ is satisfiable, then there must exist at least one trace that makes φ true, and such a trace can be used as a test. Satisfiability checking can be reduced to model checking by introducing a *universal model* M that denotes all possible traces over the set of propositions. In this model, φ is satisfiable precisely when $M \not\models \neg\varphi$, yielding an error trace, our test. A detailed study of translations from LTL to Büchi automata and their use in model checkers is provided in [23]. In [18] the authors investigate satisfiability checking for Mission-Time LTL (MLTL), a time-bounded variant of temporal logic. They develop reductions that map MLTL satisfiability into existing frameworks, including LTL satisfiability via automata-based techniques, and model checking with nuXmv. In addition, they propose an SMT-based encoding of MLTL satisfiability using Z3. The SMT encoding is similar to ours, although they use universal and existential quantification over trace positions, whereas we, for efficiency reasons, have chosen to unfold the quantifiers using conjunction and disjunction. A similar translation to SMT is pursued in [4] for MITL (Metric Interval Temporal Logic) satisfiability checking. In [28] the authors derive auxiliary sub properties from a future time propositional LTL requirement and use a model checker to generate execution scenarios (traces) for them. These scenarios are then used for monitoring the implementation, and coverage is achieved when every such scenario has been exercised by at least one observed execution.

A subject closely related to test generation from temporal formulas is planning [21,30]. E.g. the PDDL3 [10] planning language standard extends the Planning Domain Definition Language (PDDL) with a fragment of LTL over finite traces. In [29] the authors introduce LTL-RE, which combines LTL operators with regular expressions, which are translated into alternating automaton, used to restrict classical planners. However, in classical planning languages such as PDDL, actions are parameterized over a finite set of typed objects. By contrast, MaTL operates on data values from potentially infinite domains.

8 Conclusion

We have presented the `fuzz` tool for generating tests, which are sequences of commands, from formulas in the MaTL temporal logic with future and past time operators, and a match construct matching events and their carried data. We defined the syntax and semantics of MaTL, and provided a collection of example properties. We explained how formulas are translated to SMT formulas, and also how monitors are generated, which verify that generated tests are valid. Since the Z3 solver cannot be relied upon to randomize its solutions, we described an extra randomization of the tests generated by Z3. The evaluation shows that the framework applied to our example properties is practical for tests of length 100 or less. We believe that tests normally would fall in this range. However, the

approach faces a scalability challenge. Especially if there are many commands. In our case we operated with five commands. We demonstrated that string arguments to commands and constraints on these severely influence scalability in a negative direction. The format of commands, namely named data records with named fields, is very general, and many other formats can be generated from this format. This makes the approach more broadly applicable beyond applications being controlled by commands. Future work includes studying how strings can be handled more efficiently. Other SMT encodings can also be investigated, e.g. by using universal and existential quantification over trace positions instead of the use of conjunction and disjunction. It remains to be seen whether one performs drastically better than the other. Finally, the tool is currently based on the Z3 SMT solver. Future work can include also interfacing to other solvers.

Acknowledgments The research was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. We thank the reviewers for their helpful comments.

References

1. Bartocci, E., Falcone, Y., Francalanza, A., Leucker, M., Reger, G.: An introduction to runtime verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 1–23. Springer (2018)
2. Basin, D.A., Klaedtke, F., Marinovic, S., Zalinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal Methods Syst. Des. **46**(3), 262–285 (2015)
3. Bauer, A., Leucker, M.: The theory and practice of SALT. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. LNCS, vol. 6617, pp. 13–40. Springer (2011)
4. Bersani, M.M., Rossi, M., San Pietro, P.: An SMT-based approach to satisfiability checking of MITL. Inf. Comput. **245**, 72–97 (2015). `https://doi.org/10.1016/J.IC.2015.06.007`
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 209–224. USENIX (2008)
6. d'Amorim, M., Havelund, K.: Event-based runtime verification of Java programs. ACM SIGSOFT Software Engineering Notes **30**(4), 1–7 (2005)
7. Demri, S., Lazić, R.: LTL with the freeze quantifier and register automata. In: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 17–26. IEEE (2006)
8. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D., Kalus, G. (eds.) Engineering Dependable Software Systems. NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013)
9. The Fuzz test case generator. `https://github.com/havelund/fuzzing`
10. Gerevini, A., Long, D.: Plan constraints and preferences in PDDL3, the language of the deterministic part of the fifth international planning competition. In: Fifth International Planning Competition, ICAPS (2006)

11. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 213–223. ACM (2005)
12. Havelund, K., Goldberg, A.: Verify your runs. In: Verified Software: Theories, Tools, Experiments, VSTTE 2005. LNCS, vol. 4171, pp. 374–383. Springer (2008). https://doi.org/10.1007/978-3-540-69149-5_40
13. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. Formal Methods in System Design **56**(1-3), 1–21 (2020)
14. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 61–102. Springer (2018)
15. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. Journal of the ACM **32**(1), 137–161 (1985). https://doi.org/10.1145/2455.2460
16. Lark - a parsing toolkit for Python. https://github.com/lark-parser/lark
17. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming **78**(5), 293–303 (may/june 2009). https://doi.org/10.1016/j.jlap.2008.08.004
18. Li, J., Vardi, M.Y., Rozier, K.Y.: Satisfiability checking for mission-time LTL. In: Computer Aided Verification. LNCS, vol. 11562, pp. 3–22. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_1
19. Milner, R.: A modal characterisation of observable machine-behaviour. In: CAAP 1981: Colloquium on Trees in Algebra and Programming. Lecture Notes in Computer Science, vol. 112, pp. 25–34. Springer (1981)
20. Peleska, J., Vorobev, I., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: NASA Formal Methods (NFM 2011). Lecture Notes in Computer Science, vol. 6617, pp. 298–312. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_22
21. Planning.Wiki. https://planning.wiki
22. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE Computer Society (1977)
23. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. International Journal on Software Tools for Technology Transfer **12**, 123–137 (2010). https://doi.org/10.1007/s10009-010-0140-3, an earlier version appeared in SPIN'07
24. Scala Programming Language. http://www.scala-lang.org
25. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference (ESEC/FSE). pp. 263–272. ACM (2005)
26. SMT-LIB. https://smt-lib.org
27. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. In: Proc. of the 5th Int. Workshop on Runtime Verification (RV'05). vol. 144(4), pp. 109–124. Elsevier (2006)
28. Tan, L., Sokolsky, O., Lee, I.: Specification-based testing with linear temporal logic. In: Proceedings of the 2004 International Conference on Information Reuse and Integration. pp. 493–498. IEEE (2004). https://doi.org/10.1109/IRI.2004.1431509
29. Triantafillou, E., Baier, J., McIlraith, S.: A unifying framework for planning with LTL and regular expressions. In: Proceedings of the Workshop on Model-Checking and Automated Planning (MOCHAP) at ICAPS. pp. 23–31 (2015)
30. Unified planning library. https://unified-planning.readthedocs.io/en/latest
31. The Z3 SMT solver. https://github.com/z3prover/z3
32. Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: The Fuzzing Book. https://www.fuzzingbook.org