STLInspector: STL Validation with Guarantees

Hendrik Roehm[1(✉)], Thomas Heinz[1], and Eva Charlotte Mayer[2]

[1] Robert Bosch GmbH, Corporate Research, Renningen, Germany
{hendrik.roehm,thomas.heinz2}@de.bosch.com
[2] Universität Tübingen, Fachbereich Informatik, Tübingen, Germany
eva-charlotte.mayer@student.uni-tuebingen.de

Abstract. STLInspector is a tool for systematic validation of Signal Temporal Logic (STL) specifications against informal textual requirements. Its goal is to identify typical faults that occur in the process of formalizing requirements by mutating a candidate specification. STLInspector computes a series of representative signals that enables a requirements engineer to validate a candidate specification against all its mutated variants, thus achieving full mutation coverage. By visual inspection of the signals via a web-based GUI, an engineer can obtain high confidence in the correctness of the formalization – even if she is not familiar with STL. STLInspector makes the assessment of formal specifications accessible to a wide range of developers in industry, hence contributes to leveraging the use of formal specifications and computer-aided verification in industrial practice. We apply the tool to several collections of STL formulas and show its effectiveness.

Keywords: Specification validation · Temporal logic · STL · MTL · SMT · Mutation testing

1 Introduction

Recently, Signal Temporal Logic (STL) [14] became increasingly popular as a specification formalism for requirements of cyber-physical systems (CPS) [13,17] [1,5–7,9,20]. An STL specification can be thought of as a set of discrete and continuous signals that represent correct behavior of a CPS over time. Since many safety-critical industrial systems are CPS, checking correctness of their behavior is crucial. A variety of methods for checking STL specifications have been developed including signal monitoring [5,17], model-based falsification [1], and formal verification of STL specifications [20]. However, to be able to trust the testing/verification machinery, it is crucial to trust the formalization of requirements. It has been observed that industrial requirements can be fairly nontrivial, thus resulting in complex formulas that are not easily understandable [19]. If a formal specification does not conform to the corresponding natural language requirement, which is the common representation of requirements in industry today,

R. Majumdar and V. Kunčak (Eds.): CAV 2017, Part I, LNCS 10426, pp. 225–232, 2017.
DOI: 10.1007/978-3-319-63387-9_11

verification results based on the specification are useless. Therefore, our tool STLInspector addresses the problem of checking an STL specification against an informal natural language requirement involving the requirements engineer as an oracle. STLInspector provides the requirements engineer with a systematic way of validating candidate STL specifications and gives her high confidence in the correctness of the formalization.

We use the example given by Dokhanchi et al. [4] to illustrate the problem and our solution. Suppose an engineer formalizes the textual requirement.

> *"At some time in the first 30 s, the vehicle speed (vel) will go over 100* $\frac{km}{h}$ *and stay above 100* $\frac{km}{h}$ *for 20 s."*

by the STL formula

$$\varphi_c = \mathcal{F}_{[0,30]}((vel > 100) \Rightarrow \mathcal{G}_{[0,20]}(vel > 100)). \tag{1}$$

However, a test signal which is generated by STLInspector and depicted in Fig. 1 shows that φ_c does not conform to the textual requirement because the test signal satisfies φ_c but not the textual requirement. The engineer can detect the faulty specification by visual inspection of the signal which requires no knowledge of STL or temporal logics in general. Hence, specification validation becomes accessible to a wide range of developers in industry.

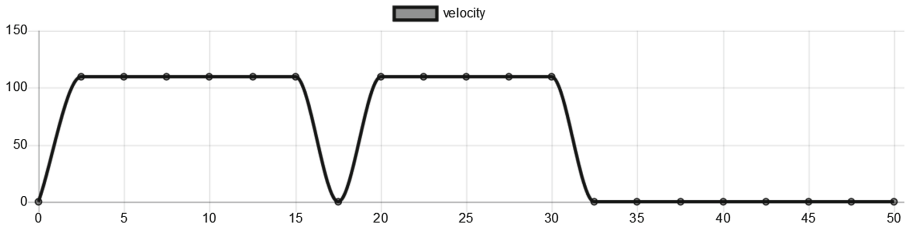

Fig. 1. A test signal – as visualized in STLInspector – that does not satisfy the textual requirement. Yet the signal satisfies its formalization φ_c, thus revealing that φ_c is incorrect.

STLInspector generates a series of such test signals that allows to show absence of typical errors made during formalization and increases confidence in its correctness. Inspired by ideas from mutation testing [3,10], typical classes of errors are formalized by mutation operators. For instance, the stuck-at-one operator produces the mutant $\varphi_c' = \mathcal{F}_{[0,30]}(true \Rightarrow \mathcal{G}_{[0,20]}(vel > 100))$ for φ_c from above. A signal is generated which does only satisfy the mutant φ_c' but not the candidate φ_c and thus represents a corner case of the formula φ_c. If the engineer identifies the behaviour as non-conforming to the textual requirement, the particular error associated with the mutation is shown to be absent. In this sense, STLInspector provides coverage guarantees for the considered set of error

classes. By adding additional mutation operators, the tool can easily be extended to also handle domain specific error classes. Signal generation is performed using an SMT encoding of STL formulas (Sect. 4). We apply the tool to several collections of STL formulas and show its effectiveness (Sect. 5). STLInspector is an academic prototype and available under Apache 2.0 license at https://github.com/STLInspector.

Related Work: Vispec [9] is a tool that provides a graphical formalism based on template patterns to formalize specifications without requiring knowledge of temporal logics. STLInspector complements Vispec by enabling validation of such formalizations. It is however not restricted to templates. Vispec was extended by Dokhanchi et al. [4] to detect validity, redundancy and vacuity in MTL formalizations. These properties can be considered simple mutations and may be incorporated in STLInspector as a special case. Mutation testing has been applied to specification validation [10, Section V.B] without considering continuous-time signals. RATSY [2] is another tool that focuses on debugging specifications via a game-based approach. In contrast to STLInspector, RATSY specifications are based on a subset of PSL (expressively equivalent to ω-regular languages). Thus, it cannot be applied to continuous-time and real-valued signals. EGRET is a similar tool for string-based specifications which generates test strings for regular expressions [12].

Signal Temporal Logic: STL was introduced by Maler and Nickovic [13,14]. A signal s is a mapping from time to the valuation of Boolean and real-valued variables. We consider bounded time signals only, i.e., $s : [0, T] \to \mathbb{B}^n \times \mathbb{R}^m$ with n Boolean variables $P = \{p_1, \ldots, p_n\}$ and m real-valued variables given by the vector $R = (r_1, \ldots, r_m)$. STL is a logic to specify temporal properties of s. It consists of Boolean variables, constraints on real-valued variables, logical and temporal operators. We focus on the linear fragment of STL and signals whose real-valued components are continuous. Its syntax is as follows.

$$\alpha := p \mid D^T R \leq e, \quad \varphi := \alpha \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{U}_I \varphi_2$$

with $p \in P$, $D \in \mathbb{R}^m$, $e \in \mathbb{R}$, and I being an interval $[a, b]$ with $a, b \in \mathbb{R}$. Note that STL semantics slightly differ across publications. We use the semantics as published in [20] and omit its definition due to space restrictions. While there exist different options to interpret unbounded time formulas over bounded time signals, for practical purposes bounded time formulas seem to be sufficient.

2 Mutation Testing and Coverage

In this section, we give a short introduction on mutation testing based on Fraser and Wotawa [8] and describe how we are able to guarantee that certain errors are not present in an STL formula φ by a set of test signals. Mutation testing involves the notion of a mutant of φ, i.e., another formula φ' which is obtained by

applying a syntactic modification to φ. For example, $\varphi'_c = \mathcal{F}_{[0,30]}((vel < 100) \Rightarrow \mathcal{G}_{[0,20]}(vel > 100))$ is a mutation of φ_c in Eq. (1) where ">" is replaced by "<" in the first constraint. This type of syntactic modification is made precise by the *relation replacement operator* (rro):

$$\begin{aligned} rro(D^T R \sim_1 e) &= \{D^T R \sim_2 e | \sim_2 \in RO, \sim_2 \neq \sim_1\} && \sim_1 \in RO \\ rro(X \star Y) &= \{x \star Y | x \in rro(X)\} \cup \{X \star y | y \in rro(Y)\} && \star \in BO \\ rro(\star\ X) &= \{\star\ x | x \in rro(X)\} && \star \in MO \end{aligned}$$

with $RO = \{\equiv, \neq, >, \leq, <\}$, $BO = \{\vee, \wedge, \rightarrow, \mathcal{U}_{[a,b]}, \mathcal{R}_{[a,b]}\}$, and $MO = \{\neg, \mathcal{F}_{[a,b]}, \mathcal{G}_{[a,b]}, \mathcal{N}_{[a]}\}$[1]. For φ_c, the relation replacement operator produces the list $rro(\varphi_c)$ of 8 mutants including φ'_c.

A signal s distinguishes φ and a mutant φ' if $s \models \varphi$ and $s \not\models \varphi'$ holds or $s \not\models \varphi$ and $s \models \varphi'$ holds. In such a case, s is said to kill the mutant φ'. For each such test signal s, the user must determine whether it conforms to the textual requirement ($\uparrow$) or not ($\downarrow$). Four cases can be distinguished.

- $s \models \varphi$, $s \not\models \varphi'$, $\uparrow$: error represented by φ' is not present in φ
- $s \models \varphi$, $s \not\models \varphi'$, $\downarrow$: φ contains illegitimate behavior
- $s \not\models \varphi$, $s \models \varphi'$, $\uparrow$: φ' contains legitimate behavior that is missing in φ
- $s \not\models \varphi$, $s \models \varphi'$, $\downarrow$: error represented by φ' is not present in φ

We consider the following mutation operators, which are adaptions of mutation operators defined by Fraser and Wotawa [8] to real-valued and continuous-time signals. In Sect. 5, we illustrate that they are in fact suitable to detect errors.

- Relation replacement
- Temporal operator insertion
- Temporal interval replacement
- Missing temporal operator
- Atomic proposition negation
- Expression negation
- Operand replacement
- Logical operator replacement
- Temporal operator replacement
- Stuck at zero
- Stuck at one
- Missing condition
- Associate shift

Due to space restrictions, we do not give additional operator definitions but refer to the documentation of STLInspector and the work by Fraser and Wotawa [8] and Mayer [15]. For a given list of mutants M, the mutation coverage of a set of signals can be defined as the percentage of mutants in M which are killed by these signals, not considering mutants that are semantically equivalent to the candidate formula. STLInspector generates sets of test signals which have 100% mutation coverage for all mutants generated by the mutation operators given above. Hence, we can guarantee that a formalization candidate does not contain any errors from a finite set of error classes where each class is

[1] Note that $\varphi\ \mathcal{R}_{[a,b]}\ \psi = \neg(\neg\varphi\ \mathcal{U}_{[a,b]}\ \neg\psi)$, $\mathcal{F}_{[a,b]}\ \varphi = \top\ \mathcal{U}_{[a,b]}\ \varphi$, $\mathcal{G}_{[a,b]}\ \varphi = \neg\mathcal{F}_{[a,b]}\ \neg\varphi$, $\mathcal{N}_{[a]}\ \varphi = \mathcal{G}_{[a,a]}\ \varphi$.

represented by a finite set of mutants. If a formula contains multiple errors, we cannot guarantee that the errors are detected unless there is a mutation operator for those specific multiple errors. However, the empirical evaluation in Sect. 5 indicates that we typically find errors also in the multiple error case. Note that the tool can be easily adapted to similar notions of coverage, for instance UFC and PICC [8].

3 Architecture of STLInspector

The tool is written in Python. STLInspector can be used as a command line tool, via the browser-based graphical user interface, or integrated into existing programs. In the following, we describe the GUI and the core components which are structured as visualized in Fig. 2.

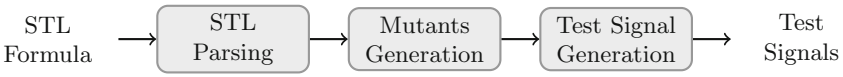

Fig. 2. Structure of the core functionality.

STL Parsing: The STL formula – written in textual form – is parsed into a syntax tree with Antlr [18]. The input format is described in the documentation. Examples are `G[1,3] vel >= 10` and `vel == 0 U[0,30] seatBeltFastened`.

Mutants Generation: In this component, all mutation operators listed in Sect. 2 are applied to the input formula. Every mutation operator outputs a list of mutants which are merged into one list containing all possible mutants.

Test Signal Generation: For a mutant φ' of the STL formula φ, STLInspector randomly chooses between the generation of a test signal s such that $s \models \varphi \wedge \neg\varphi'$ and $s \models \neg\varphi \wedge \varphi'$ to avoid bias on the satisfaction of φ. A test signal s is generated using the SMT encoding described in Sect. 4. Furthermore, it is checked whether the test signal s can be used to kill additional mutants. Test signal generation is repeatedly performed until a set S of test signals is obtained such that every mutant – except equivalent mutants – is killed by at least one element of S. Note that one test signal typically kills multiple mutants, thus less test signals are required than mutants (Sect. 5).

Web-Based GUI: STLInspector includes a front-end similar to Jupyter [11]. The user can enter an STL formula and the corresponding informal textual requirement. The front-end shows the generated test signals and the user decides whether or not the signal satisfies the informal requirement. STLInspector outputs which one of the four cases of Sect. 2 applies. If an error was found, the

user can change the STL candidate and continue the visual inspection. For one STL candidate, the evaluation results of different users can be saved and easily compared on the project overview page.

4 Test Signal Encoding and Generation

For a given STL formula $\overline{\varphi}$, a test signal s which satisfies $s \models \overline{\varphi}$ is generated using the SMT-solver Z3 [16]. In the following, the SMT encoding of $s \models \overline{\varphi}$ is sketched. Time is partitioned into an alternating sequence of points and open intervals, similar to Maler and Ničković [13], however with a fixed time step c:

$$I_T = \{\{0\}, (0,c), \{c\}, (c,2c), \ldots\}, \quad [0,T] = \bigcup\nolimits_{I \in I_T} I$$

The parameters c and T are selected automatically where c must divide T and all bounds of temporal operator intervals in the formula $\overline{\varphi}$. Signals are generated such that the value of Boolean variables is constant for intervals in I_T. The set of such signals satisfying a formula can be encoded as an SMT formula using the rewriting technique by Roehm et al. [20]. For instance, the formula $\overline{\varphi}_c = \overline{\varphi}_1 \mathcal{U}_{[0,1]} \overline{\varphi}_2$ can be rewritten as follows:

$$\begin{aligned} s \models \overline{\varphi}_c &\Leftrightarrow s \models (\overline{\varphi}_1 \mathcal{U}_{[0,0]} \overline{\varphi}_2) \vee (\overline{\varphi}_1 \mathcal{U}_{(0,1)} \overline{\varphi}_2) \vee (\overline{\varphi}_1 \mathcal{U}_{[1,1]} \overline{\varphi}_2) \\ &\Leftarrow s \models \overline{\varphi}_2 \vee (\overline{\varphi}_1 \wedge \mathcal{G}_{(0,1)} \overline{\varphi}_1 \wedge \mathcal{F}_{(0,1)} \overline{\varphi}_2) \vee (\overline{\varphi}_1 \wedge \mathcal{G}_{(0,1)} \overline{\varphi}_1 \wedge \mathcal{F}_{[1,1]} \overline{\varphi}_2) \end{aligned}$$

The rewritten formula can be expressed by the SMT formula $\overline{\varphi}_2^0 \vee (\overline{\varphi}_1^0 \wedge \overline{\varphi}_1^{0.5} \wedge \overline{\varphi}_2^{0.5}) \vee (\overline{\varphi}_1^0 \wedge \overline{\varphi}_1^{0.5} \wedge \overline{\varphi}_2^1)$ using $\overline{\varphi}_1^0 = \overline{\varphi}_1$, $\overline{\varphi}_1^{0.5} = \mathcal{G}_{(0,1)} \overline{\varphi}_1$, etc., and solved by Z3 [16]. The encoding ensures that for a real variable, the continuous signal obtained from piecewise linear interpolation of the sample points satisfies $\overline{\varphi}$ for the linear fragment of STL. The full theory [15] is omitted due to space restrictions.

5 Evaluation

We evaluate the effectiveness of mutation-based test signals in finding errors by two case studies. First, we use STLInspector to check STL formulas published as part of the UnCoVerCPS EU project [21]. They identified 8 common requirement patterns and formalized them by STL formulas and timed monitor automata. Since the patterns contain unbounded operators, we replace them by bounded ones. For the 4th requirement, one signal shows that the bounded STL formula does not conform to the requirement. Furthermore, the same signal shows that the original unbounded STL formula (as well as the monitor automaton) does not conform to the requirement either[2]. Our second evaluation is based on data of an online survey[3] by Dokhanchi et al. [4]. They requested participants to

[2] In fact, the proposed formula is equivalent to $\mathcal{G}_{[0,\infty)}(q \Rightarrow \mathcal{G}_{[0,\infty)}(p \Rightarrow \mathcal{G}_{[0,\infty)}p))$, which formalizes "after q, once p becomes true, p holds forever".

[3] We gratefully acknowledge the support of Bardh Hoxha and his colleagues to get access to some results of their survey.

write STL formalizations for several informal textual requirements. For each of the 66 formalizations φ_i, that we have access to from the survey, we generate a set S_i of test signals with 100% mutation coverage. For each formalization, STLInspector generates 6 test signals on average (minimum 3, maximum 11). We check whether φ_i can be distinguished from the correct formalization φ_c based on the test signals of S_i. Out of the 66 formalizations with 31 being unique, we are able to distinguish all of the 44 faulty ones (26 unique ones) from the correct formalizations. Since we are able to detect all faulty formalizations with our test generation, our list of mutation operators is sufficient to detect errors for the given formalizations. Since 12 of the 26 unique faulty formalizations need more than one mutation to transform them into the correct formula, we are able to discover the faulty formalizations even in the case where we do not have a guarantee to do so. We conclude from both case studies that mutation-based specification validation with STLInspector helps in finding errors and increasing confidence in correctness of STL formalizations.

References

1. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19835-9_21
2. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY – A new requirements analysis tool with synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 425–429. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14295-6_37
3. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178–187 (1978)
4. Dokhanchi, A., Hoxha, B., Fainekos, G.E.: Metric interval temporal logic specification elicitation and debugging. In: MEMOCODE 2015, pp. 70–79 (2015)
5. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 264–279. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_19
6. Dreossi, T., Dang, T., Donzé, A., Kapinski, J., Jin, X., Deshmukh, J.V.: Efficient guiding strategies for testing of temporal properties of hybrid systems. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 127–142. Springer, Cham (2015). doi:10.1007/978-3-319-17524-9_10
7. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoret. Comput. Sci. **410**(42), 4262–4291 (2009)
8. Fraser, G., Wotawa, F.: Complementary criteria for testing temporal logic properties. In: Dubois, C. (ed.) TAP 2009. LNCS, vol. 5668, pp. 58–73. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02949-3_6
9. Hoxha, B., Mavridis, N., Fainekos, G.E.: VISPEC: A graphical tool for elicitation of MTL requirements. In: IROS 2015, Hamburg, Germany, 28 September–2 October 2015, pp. 3486–3492 (2015)
10. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)

11. Kluyver, T., et al.: Jupyter notebooks - a publishing format for reproducible computational workflows. In: Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, 7–9 June 2016, pp. 87–90 (2016)
12. Larson, E., Kirk, A.: Generating evil test strings for regular expressions. In: ICST 2016, Chicago, IL, USA, 11–15 April 2016, pp. 309–319 (2016)
13. Maler, O., Ničković, D.: Monitoring properties of analog and mixed-signal circuits. Int. J. Softw. Tools Technol. Transfer **15**, 247–268 (2013)
14. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT 2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30206-3_12
15. Mayer, E.C.: Mutation-based validation of temporal logic specifications with guarantees. Bachelor's thesis, Universität Tübingen (2017)
16. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
17. Nickovic, D., Maler, O.: AMT: A property-based monitoring tool for analog systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 304–319. Springer, Heidelberg (2007). doi:10.1007/978-3-540-75454-1_22
18. Parr, T.: The Definitive ANTLR 4 Reference, 2nd edn. Pragmatic Bookshelf, Raleigh (2013)
19. Roehm, H., Gmehlich, R., Heinz, T., Oehlerking, J., Woehrle, M.: Industrial examples of formal specifications for test case generation. In: ARCH@CPSWeek 2015, Seattle, WA, USA, pp. 80–88 (2015)
20. Roehm, H., Oehlerking, J., Heinz, T., Althoff, M.: STL model checking of continuous and hybrid systems. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 412–427. Springer, Cham (2016). doi:10.1007/978-3-319-46520-3_26
21. Schuler, S., Walsch, A., Woehrle, M.: Deliverable D1.1 - Assessment of languages and tools for the automatic formalisation of system requirements. Technical report, as part of the EU-Project UnCoverCPS (2015). http://cps-vo.org/node/24197