Towards Strengthening Formal Specifications with Mutation Model Checking

Maxime Cordy
SnT, University of Luxembourg
Luxembourg

Sami Lazreg
SnT, University of Luxembourg
Luxembourg

Axel Legay
Université Catholique de Louvain
Belgium

Pierre Yves Schobbens
University of Namur
Belgium

ABSTRACT

We propose mutation model checking as an approach to strengthen formal specifications used for model checking. Inspired by mutation testing, our approach concludes that specifications are not strong enough if they fail to detect faults in purposely mutated models. Our preliminary experiments on two case studies confirm the relevance of the problem: their specification can only detect 40% and 60% of randomly generated mutants. As a result, we propose a framework to strengthen the original specification, such that the original model satisfies the strengthened specification but the mutants do not.

CCS CONCEPTS

• **Software and its engineering → Model checking**; • **Theory of computation → Logic and verification**.

KEYWORDS

Model checking, LTL, Mutation

ACM Reference Format:
Maxime Cordy, Sami Lazreg, Axel Legay, and Pierre Yves Schobbens. 2023. Towards Strengthening Formal Specifications with Mutation Model Checking. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3611643.3613080

1 CONTEXT, MOTIVATIONS AND AIMS

Model checking [9] exhaustively assesses whether the behaviour of a given system (captured in a formal model) satisfies its formal specification (expressed, e.g., in temporal logic like Linear Temporal Logic (LTL) [32]). The guarantees that model checking offers are bound to i) the adequacy of the model to the real system and ii) the soundness and strength of the specification. Soundness refers to whether the specification captures only properties that the real system should satisfy, whereas its strength refers to the extent to which the specification forbids undesired behaviour. While past research has approached the problems of building an adequate system model [4, 10] and synthesizing *sound* specifications [14, 18], the problem of specifying strong specifications remains open and challenging to address. This gap constitutes a serious pitfall for model checking: weak specifications will fail to detect undesired behaviour and, in turn, reduce the trust engineers can have in model-checking results.

In this vision paper, we present a new endeavour toward automatically strengthening specification in model checking. Our idea takes inspiration from *mutation testing*, a software testing technique used to evaluate the strength of test suites and generate new test cases for uncovered behaviour. The principled hypothesis of mutation testing is that, given an executable piece of software code, strong test suites should detect (fails on) any modification of the software code that is not semantic-preserving. Hence, mutation testing operates multiple alterations of the original source code to produce so-called "mutants", and evaluates test suite strength as the ratio of mutants that the test suite "kills" (i.e., mutants that make the test suite fail). Then, the test suite can be augmented with new test cases that fail on the "surviving" mutants.

Our approach – which we name *"mutation model checking"* – applies the principles of mutation testing to assess and improve the strength of formal specifications. Starting from a (supposedly correct) model and its specification, we generate multiple mutants of the model. If some mutant satisfies the specification, our approach synthesizes a novel logic formula that can distinguish the mutant from the original model, i.e., which is violated by the mutant but still satisfied by the original model. This new formula can be added to (i.e., strengthen) the specification. The process can then be repeated until no surviving mutant can be generated.

We demonstrate that existing academic exemplars and industrial cases of model checking suffer from weak specifications. We randomly mutate Promela[17] models and reveal that a large portion (38% and 60%) of syntactically correct mutants (non-equivalent to the original model) satisfy the companion LTL specification. In an attempt to solve this issue, we outline the "mutation model checking" approach and present the first sketches of a solution framework. We describe the framework component and highlight the research challenges that have to be solved for this framework to be fully operational.

2 RELATED WORK

Mutation testing has been applied to various contexts and formalisms [28]. Mutation testing has already been combined with model checking for guiding test-case generation [1–3, 6, 20, 30, 33].

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613080

Mutants of the formal model are used to generate traces that are not producible by the original (so-called negative traces) model to generate new test cases for the source program. Consequently, it can also be used to assess the quality of a program test suite. Similarly, when mutants represent potential system vulnerabilities, negative traces that do not satisfy the properties can be used to demonstrate system defects [33]. Mutation testing in a formal verification context has also been used to validate the specification [30] rather than generate tests. However, none of these approaches proposes to strengthen the specification automatically.

3 PRELIMINARY EXPERIMENTS

3.1 Case Studies

3.1.1 Minepump. Minepump [11, 12, 19] is a well-known academic exemplar that mimics the behaviour of a minepump software controller. This software implements the control logic of the minepump according to sensors (e.g. methane and water) and pump actuator states. We have retrieved a model of the minepump written in the Promela language [17], which contains about 300,000 states and its companion LTL specification includes 12 properties.

3.1.2 ADAPRO. ADAPRO stands for "ALICE" Data Point Processing Framework. ALICE is one of the four major experiments at the Large Hadron Collider (LHC) of the European Organisation for Nuclear Research (CERN). The main focus of the framework (about 6000 lines of codes written in C++) is on producing multi-threaded applications with optional support for configuration files and remote control and monitoring[1].

The highly concurrent nature of ADAPRO makes its validation challenging by conventional testing. Although the framework and its applications [21] have been subject to hundreds of hours of unit testing, integration testing, and acceptance testing, ADAPRO is too complex for all its behaviours to be anticipated by tests. LTL model checking has been applied to model ADAPRO in Promela and verify 17 critical properties [22].[2] Model checking managed to reveal important issues not previously found by testing. The findings necessitated changes in the design and implementation of ADAPRO [22].

3.2 Objectives and Experimental Protocol

Our preliminary experiments aim to identify weaknesses in the LTL specifications of the two case studies. To achieve this, we take inspiration from *mutation testing* and randomly apply predefined mutation operators to alter the Promela models. We, then, assess the models against the existing specifications and compute the *mutation score*, i.e. the percentage of mutants that do not satisfy the specification. This percentage is a proxy that we use to measure the strength of the specification.

3.2.1 Mutant generation. We generate mutants from original Promela models by applying syntactic mutations of the Promela code (just like code statements are altered in classical mutation testing). More precisely, we have developed a tool that mutates the abstract syntax tree (AST) representation of the Promela code. Any Promela literal and operator can be changed into another terminal/operator of the same type. For example, a "+" arithmetic node has {"-", "/", "*", "%"} as possible mutations. Mutants are generated uniformly by mutating each relevant AST node once, except those found in statements that have, by design, no impact on behaviour (e.g. print statements) – this, however, does not entirely prevents the equivalent mutant problem (read more in Section 4.3). This results in 88 mutants for the Minepump case study and 406 for ADAPRO, as this corresponds to the number of terminal AST nodes.

3.2.2 Mutant analysis. We, then, run the SPIN model checker [17] to check every generated mutant against the specification. We record the mutation score for each LTL formula that composes the specification, and the mutation score of the full specification (i.e. the conjunction of all formulae).

3.3 Results

Table 1 presents the results for each case study. The total mutation score of Minepump is close to 60%, and that of ADAPRO barely reaches 40%. These results demonstrate that, surprisingly, existing specifications may fail to capture a significant number of faults introduced in the models.

3.3.1 Minepump. Interestingly, the formulae that kill the most mutants (#1, #7 and #8) have a mutation score (around 30%) that is still far from the total mutation score (60%). This indicates a certain lack of redundancy within the set of formulae, and the absence of one-size-fits-all solutions to kill all mutants. Other formulae (#4 and #5) kill no mutant, indicating that they model either trivial properties (and should therefore be reconsidered) or intricate properties that only much different models could violate. Nevertheless, the fact that a significant number of mutants survive indicates that the need for strengthening the specification.

3.3.2 ADAPRO. Compared to Minepump, there is a smaller difference in mutation scores between the full specification and the individual formulae. This shows that not only the specification is insufficient, but also the individual formulae cover redundant faults in the model. Thus, there is a large part of the model behaviour that none of the formulae covers. These results confirm that available model-checking case studies actually rely on specifications with apparent weaknesses.

4 MUTATION MODEL CHECKING

4.1 Problem Formulation

Let $\Phi = \phi_1 \wedge \cdots \wedge \phi_n$ be a formal specification expressed in LTL and $\mathcal{M}$ be a system model such that $\mathcal{M} \models \Phi$, or equivalently that $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\Phi)$ where $\mathcal{L}(.)$ denotes the trace language executable by a model or accepted by a specification.

Let $\mathcal{M}'$ be a model mutant, such that $\mathcal{L}(\mathcal{M}') \setminus \mathcal{L}(\mathcal{M}) \neq \emptyset$ (i.e. $\mathcal{M}$ does not subsumes $\mathcal{M}'$).[3] We look for surviving mutants, i.e. $\mathcal{M}' \models \Phi$.

Mutation model checking is the problem of generating such mutants $\mathcal{M}'$ and synthesizing a specification Φ' such that (i) $\mathcal{L}(\Phi') \subset \mathcal{L}(\Phi)$ (or equivalently, $\Phi' \implies \Phi$), that is, Φ' *strengthens* Φ; (ii) $\mathcal{M} \models \Phi'$; (iii) $\mathcal{M}' \not\models \Phi'$. This definition can be trivially generalized

[1] https://gitlab.cern.ch/adapos/adapro
[2] https://gitlab.com/jllang/adapro/tree/5.0.0-RC3/models/promela

[3] Otherwise, $\mathcal{M}'$ has no trace that M does not have, which implies that no LTL formula that $\mathcal{M}$ satisfies can kill $\mathcal{M}'$.

Table 1: Models and LTL formulae used in our experiments.

	Minepump, Total Mutation Score = 61.458% (88 mutants)	
#1	$\Box(pumpOn \implies stateRunning)$	34.375%
#2	$\Box(readCommand \vee readAlarm \vee readLevel)$	15.625%
#3	$\Box(stateReady \vee stateRunning \vee stateStopped \vee stateMethanestop \vee stateLowstop)$	20.833%
#4	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow \Box((pumpOn \wedge methane) \Rightarrow \Diamond\neg pumpOn)$	0%
#5	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow \neg\Diamond\Box(pumpOn \wedge methane)$	0%
#6	$\Box((\neg pumpOn \wedge methane \wedge \Diamond\neg methane) \Rightarrow ((\neg pumpOn) \cup \neg methane))$	13.541%
#7	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow (\Box(lowWater \Rightarrow (\Diamond\neg pumpOn)))$	31.250%
#8	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow (\neg\Box\Diamond(pumpOn \wedge lowWater))$	31.250%
#9	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow (\neg\Diamond\Box(\neg pumpOn \wedge \neg methane \wedge highWater))$	11.458%
#10	$\Box((\neg pumpOn \wedge lowWater \wedge \Diamond highWater) \Rightarrow ((\neg pumpOn) \cup highWater))$	11.458%
#11	$\Box(lowWater \vee mediumWater \vee highWater)$	22.916%
#12	$\Box(userStart \vee userStop)$	11.458%
	ADAPRO, Total Mutation Score = 39.803%, (406 mutants)	
#1	$\forall t \in Thread, null(t) \cup (ready(t)\ \mathrm{W}\ starting(t))$	7.371%
#2	$\forall t \in Thread, starting(t) \implies (starting(t) \cup (running(t) \vee paused(t) \vee halting(t)))$	31.203%
#3	$\forall t \in Thread, running(t) \implies (running(t)\ \mathrm{W}\ (paused(t) \vee halting(t)))$	21.867%
#4	$\forall t \in Thread, paused(t) \implies (paused(t)\ \mathrm{W}\ (running(t) \vee stopping(t)))$	21.867%
#5	$\forall t \in Thread, stopping(t) \implies (stopping(t) \cup (stopped(t) \vee aborting(t)))$	24.815%
#6	$\forall t \in Thread, stopped(t) \implies stopped(t)$	21.867%
#7	$\forall t \in Thread, aborting(t) \implies (aborting(t) \cup aborted(t))$	21.867%
#8	$\Box(\forall t \in Thread, aborted(t) \implies \Box aborted(t))$	21.867%
#9	$\Box(ready(supervisor) \implies \forall v \in \{Thread \setminus supervisor\}, null(v))$	21.867%
#10	$\Box(halted(supervisor) \implies \forall v \in \{Thread \setminus supervisor\}, null(v))$	21.867%
#11	$\Box(\exists v \in \{Thread \setminus supervisor\}, ready(v) \implies starting(supervisor))$	21.867%
#12	$\Box(\forall v \in \{Thread \setminus supervisor\}, stopped(v) \implies (paused(supervisor) \vee stopped(supervisor)))$	29.238%
#13	$\Box(\exists v \in \{Thread \setminus supervisor\}, aborted(v) \implies (paused(supervisor) \vee stopped(supervisor)))$	36.363%
#14	$\Box(halting(supervisor) \implies \Diamond\forall v \in \{Thread \setminus supervisor\}, halted(v))$	22.113%
#15	$\neg\Diamond aborted(supervisor)$	24.324%
#16	$\forall t \in Thread, \Diamond\Box(executable(t) \implies \Diamond executing(t))$	25.552%
#17	$\forall t \in Thread, \Box\Diamond(executable(t) \implies executing(t))\ \mathrm{W}\ halting(supervisor)$	24.570%

to any number of mutants. One can also express the problem in *additive form*, meaning that $\Phi' \equiv \Phi \wedge \phi'$. It then boils down to generate a single LTL formula ϕ' such that $\mathcal{M} \models \phi'$ and $\mathcal{M}' \not\models \phi'$.

4.2 Framework

Figure 1 shows the four steps and components of the mutation model checking process. Starting from $\mathcal{M}$ and Φ, our framework 1) generates model mutants. These mutations occur by applying predefined mutation operators either on the modelling language (Promela here) or directly on the underlying automaton. Then, 2) each mutant $\mathcal{M}'$ is assessed against Φ using a standard model checking tool (here SPIN). Mutants that do not satisfy Φ are labelled as "killed" and the remaining ones as "surviving". If there are surviving mutants, 3) the framework generates *positive traces* (which the new specification should accept, i.e. traces of the original model) and *negative traces* (which the new specification should reject, i.e. traces of the mutant(s) that the original model cannot execute). Finally, based on these traces the framework 4) synthesizes a novel specification that can distinguish the mutant from the original model – either by updating Φ into Φ' or more simply by generating a new formula ϕ' to conjunct with Φ. The process can then be repeated until no surviving mutant can be generated.

4.3 Future Plans

To turn our work into a full research contribution, our future plans involve the implementation of our framework and its thorough evaluation based on model and specifications available in the literature, which exhibit varying characteristics (including size). We plan to focus firstly on Promela model and LTL specifications. We will later consider extending the framework to other modelling languages and logic (e.g. NuSMV [29] and CTL). To make our framework fully operational, however, we have to solve fundamental research challenges that are either inherent to any mutation analysis approach or are due to the specific settings of model checking.

4.3.1 Quality in mutant generation. The fact that a large number of mutants can be generated from a single model constitutes a challenge for any mutation analysis approach – and mutation model checking is no exception. We, therefore, have to revisit past heuristics proposed to reduce the number of generated mutants to "quality" (i.e. useful) mutants [31].

One challenge lies in the large number of *equivalent mutants*, i.e. mutants that have the same semantic behaviour as the original model. In our preliminary experiments, we have approached this issue partially, by mutating only nodes that may affect the model

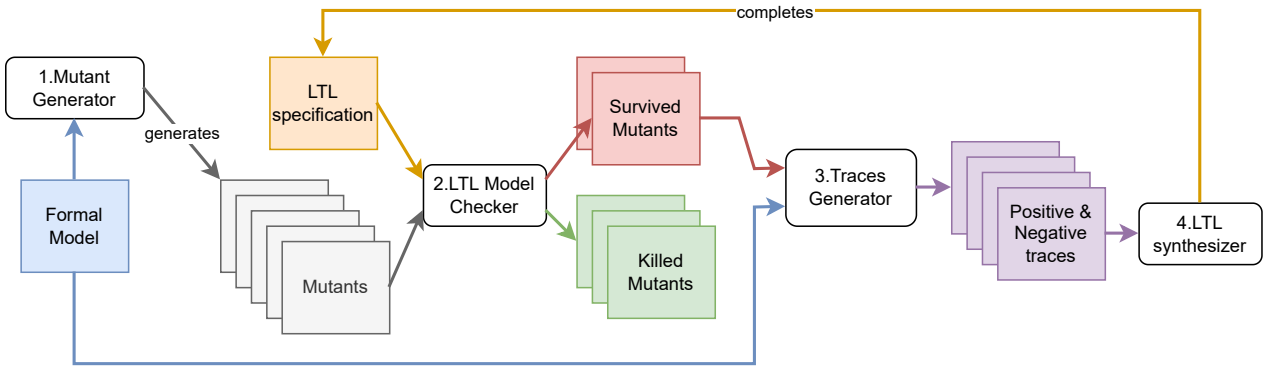

Figure 1: Framework solution for mutation model checking

execution traces (e.g. we have ignored print statements). Although this was enough to demonstrate the existence of the problem, a full-fledged solution should address the pernicious form that the equivalent mutant problem takes in mutation model checking.

Since we target LTL specifications (which universally quantifies properties over sets of traces), the concept of equivalent mutant is generalized to that of subsumed mutant: the original model cannot subsume a mutant. Otherwise, no LTL formula can kill this mutant. Model subsumption can be assessed through simulation relation [5], which we plan to complement with syntactic heuristics.

In general, two models may have non-equivalent sets of traces but still both implement the desired high-level behaviour. This can happen if the models differ in implementation details. This raises the need for project mutant assessment onto properties that reflect only high-level functional behaviour. Concretely, this means that two mutants should be considered equivalent/subsumed if they differ only in details that the specification should not consider.

4.3.2 Scalability in mutant assessment. Model-checking each mutant against every formula of the specification becomes expensive as the number of mutants and formulae grow. Moreover, the number of mutants may also affect the number of negative traces to get before formula synthesis. There is, therefore, room for exploring how to factorize our framework's execution across several mutants and formulae. To do so, we plan to develop a mutant analysis approach based on variability and statistical model checking [13, 24].

4.3.3 Formula synthesis. Synthesizing formulas from sets of solutions is a hot topic in the non-temporal case [8, 16, 23]. In the temporal case, most of the work has been dedicated to learning automata representing sets of solutions [7, 26].

Synthesizing LTL formulae is a more challenging problem. We are interested in finding a temporal formula that is satisfied by the positive traces but not by the negative ones. However, existing approaches [15, 25, 27] consider positive traces only and need a predetermined formula template, which requires a-priori knowledge of the formula to synthesize. Because we consider many mutants, we do not have this knowledge at our disposal.

4.3.4 Process Utility. To be useful, our mutation model checking process has to provide engineers with actionable results. One key challenge in this regard is the need to focus the strengthening process of the specification to the model behaviour that matters. We aim at a refinement of the specification, which should not capture all implementation details of the model. The mutation model checking process should preserve the delicate balance between specification strength and the abstraction that the specification offers over the model implementation.

Another challenge is to support useful interactions with engineers to improve their understanding and enable their participation in the refinement of the specification. Because the process inherently produces a significant number of faults (mutants) and candidate specification, we believe this opens a dual opportunity to (i) empower engineers with a better understanding of the system's specification, model and later implementation and (ii) use them as an additional guidance during specification synthesis. This requires producing intelligible representations of specifications and their semantics (expressed in authorized/forbidden system traces). This perspective can bring significant benefit at a later stage of our research and we will properly study it once our implementation of the mutation model checking process is complete.

5 CONCLUSION

In this vision paper, we have demonstrated – on two well-known case studies from the scientific literature – that existing specifications are often not strong enough, as they are also satisfied by faulty models. As a result, we formalized mutation model checking as the problem of strengthening a specification helped by mutation analysis. We have proposed a framework solution that we plan to implement in the future. Instantiating the framework, however, raises important research challenges that we have to overcome. These challenges concern the need for dedicated formula synthesis algorithms and the general scalability of the framework (due to the number of mutants, the growing size of the specification, and the number of iterations).

ACKNOWLEDGEMENT

Maxime Cordy and Sami Lazreg are supported by FNR Luxembourg (INTER/FNRS/20/15077233/Scaling Up Variability/Cordy). Axel Legay and Pierre-Yves are supported by FNRS Belgium (respectively, grants PDR/PDN - T013721 and T019921F).

REFERENCES

[1] Bernhard K Aichernig. 2003. Mutation testing in the refinement calculus. *Formal Aspects of Computing* 15, 2 (2003), 280–295.

[2] Bernhard K Aichernig. 2013. Model-based mutation testing of reactive systems. In *Theories of Programming and Formal Methods*. Springer, 23–36.

[3] Paul E Ammann, Paul E Black, and William Majurski. 1998. Using model checking to generate tests from specifications. In *Proceedings second international conference on formal engineering methods (Cat. No. 98EX241)*. IEEE, 46–54.

[4] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press.

[5] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

[6] Timothy A Budd and Ajei S Gopal. 1985. Program testing by specification mutation. *Computer languages* 10, 1 (1985), 63–73.

[7] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. 2009. Learning Minimal Separating DFA's for Compositional Verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5505)*, Stefan Kowalewski and Anna Philippou (Eds.). Springer, 31–45. https://doi.org/10.1007/978-3-642-00768-2_3

[8] Yu-Fang Chen and Bow-Yaw Wang. 2013. BULL: A Library for Learning Algorithms of Boolean Functions. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 537–542. https://doi.org/10.1007/978-3-642-36742-7_38

[9] Edmund M Clarke, Orna Grumberg, and David E Long. 1994. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1512–1542.

[10] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8

[11] Andreas Classen. 2010. Modelling with FTS: a collection of illustrative examples. *PReCISE Research Center, University of Namur, Namur, Belgium, Tech. Rep. P-CS-TR SPLMC-00000001* (2010).

[12] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and their Application to LTL Model Checking. *Transactions on Software Engineering* (2013), 1069–1089.

[13] Maxime Cordy, Sami Lazreg, Mike Papadakis, and Axel Legay. 2021. Statistical model checking for variability-intensive systems: applications to bug detection and minimization. *Formal Aspects Comput.* 33, 6 (2021), 1147–1172. https://doi.org/10.1007/s00165-021-00563-2

[14] Uli Fahrenberg, Axel Legay, and Louis-Marie Traonouez. 2014. Specification Theories for Probabilistic and Real-Time Systems. In *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8415)*, Saddek Bensalem, Yassine Lakhnech, and Axel Legay (Eds.). Springer, 98–117. https://doi.org/10.1007/978-3-642-54848-2_7

[15] Mark Gabel and Zhendong Su. 2008. Symbolic mining of temporal specifications. In *Proceedings of the 30th international conference on Software engineering*. 51–60.

[16] Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. 2020. Manthan: A Data-Driven Approach for Boolean Function Synthesis. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12225)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 611–633. https://doi.org/10.1007/978-3-030-53291-8_31

[17] G. J. Holzmann. 2004. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley.

[18] Paul Hunter, Guillermo A. Pérez, and Jean-François Raskin. 2022. Correction to: Reactive synthesis without regret. *Acta Informatica* 59, 5 (2022), 671. https://doi.org/10.1007/s00236-021-00410-0

[19] J. Kramer, J. Magee, M. Sloman, and A. Lister. 1983. CONIC: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E* 130, 1 (1983), 1–10.

[20] Willibald Krenn and Bernhard K Aichernig. 2009. Test case generation by contract mutation in spec. *Electronic Notes in Theoretical Computer Science* 253, 2 (2009), 71–86.

[21] J Lång et al. 2017. ADAPOS: An Architecture for Publishing ALICE DCS Conditions Data. In *Proc. of International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17), Barcelona, Spain*. 8–13.

[22] John Lång and ISWB Prasetya. 2019. Model checking a C++ software framework: a case study. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1026–1036.

[23] Louis Latour. 2004. From Automata to Formulas: Convex Integer Polyhedra. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*. IEEE Computer Society, 120–129. https://doi.org/10.1109/LICS.2004.1319606

[24] Axel Legay, Anna Lukina, Louis-Marie Traonouez, Junxing Yang, Scott A. Smolka, and Radu Grosu. 2019. Statistical Model Checking. In *Computing and Software Science - State of the Art and Perspectives*, Bernhard Steffen and Gerhard J. Woeginger (Eds.). Lecture Notes in Computer Science, Vol. 10000. Springer, 478–504. https://doi.org/10.1007/978-3-319-91908-9_23

[25] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL specification mining (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 81–92.

[26] Yong Li, Yu-Fang Chen, Lijun Zhang, and Depeng Liu. 2021. A novel learning algorithm for Büchi automata based on family of DFAs and classification trees. *Inf. Comput.* 281 (2021), 104678. https://doi.org/10.1016/j.ic.2020.104678

[27] Daniel Neider and Ivan Gavran. 2018. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–10.

[28] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[29] Malte Plath and Mark Ryan. 2001. Feature integration using a feature construct. *SCP* 41, 1 (2001), 53–84.

[30] Thitima Srivatanakul, John A Clark, Susan Stepney, and Fiona Polack. 2003. Challenging formal specifications by mutation: a CSP security example. In *Tenth Asia-Pacific Software Engineering Conference, 2003.* IEEE, 340–350.

[31] Chang-Ai Sun, Feifei Xue, Huai Liu, and Xiangyu Zhang. 2017. A path-aware approach to mutant reduction in mutation testing. *Inf. Softw. Technol.* 81 (2017), 65–81. https://doi.org/10.1016/j.infsof.2016.02.006

[32] Moshe Y. Vardi and Pierre Wolper. 1986. An automata-theoretic approach to automatic program verification. In *LICS'86*. IEEE CS, 332–344.

[33] Guido Wimmel and Jan Jürjens. 2002. Specification-based test generation for security-critical systems using mutations. In *International Conference on Formal Engineering Methods*. Springer, 471–482.

Received 2023-05-03; accepted 2023-07-19