TLTk: A Toolbox for Parallel Robustness Computation of Temporal Logic Specifications Extended Report

Joseph Cralley [1], Ourania Spantidi [1], Bardh Hoxha [2], and Georgios Fainekos [3]

[1] Southern Illinois University, Carbondale IL 62901
{jkolecr, ourania.spantidi}@siu.edu
[2] Toyota Research Institute North America, Ann Arbor, MI 48105
bardh.hoxha@toyota.com
[3] Arizona State University, Tempe, AZ 85281
fainekos@asu.edu

Abstract. This paper presents the Temporal Logic Toolkit (TLTk), a modular falsification tool for signal temporal logic specifications developed in Python and C. At the core of the tool, an algorithm for robustness computation is utilized that supports multi-threaded CPU/GPU computation. The tool enables parallel robustness computation of large traces. In addition, the python implementation enables the addition and modification of temporal operators for application-specific scenarios. The performance of the tool is evaluated against state-of-the-art robustness computation engines DP-TaLiRo and Breach.

Keywords: Testing · Temporal Logic · Robustness.

1 Introduction

The theory of the robustness of temporal logics has been utilized in a wide-array of problems, from testing and verification of Cyber-Physical Systems (CPS) to monitoring and planning for autonomous systems [9,6,21,11]. It enables the formulation of the falsification problem [18,15], i.e. the problem of finding system behaviors that do not meet system requirements, as a non-convex, non-linear optimization problem.

Falsification, and the related problem of parameter mining [13,14,12], have been used successfully for testing industrial-size systems. In each optimization loop in the falsification process, the two main computational elements are the system simulator and the robustness computation engine. To improve this process, we introduce TLTk[4], a Python/C toolkit for requirements-based testing of CPS. TLTk is developed with the goal of optimizing the robustness computation

[4] The source code for TLTk is publicly available through the `git` repository: https://bitbucket.org/versyslab/tltk/. Docker image: https://hub.docker.com/r/bardhh/tltk. Python package through PyPi: https://pypi.org/project/tltk-mtl/. User Guide: http://www.bhoxha.com/tltk.

engine as much as possible. At the core of the tool, a robustness computation engine that supports multi-threaded CPU and GPU computations is utilized. The memory-efficient algorithm enables robustness computations of large system traces. In addition, the robustness algorithm written in Python/C allows for easy modification/addition of temporal logic operators for application-specific implementations. This is particularly useful in areas such as planning for robotic applications since notions of robustness are usually application-specific.

TLTk supports stand-alone *Falsification* for STL/MTL specifications. Also, we provide a repository through the OS-virtualization engine Docker that allows easy integration with other tools or deployment in large-scale cloud systems like Amazon AWS, Google Cloud or Microsoft Azure. TLTk has been successfully utilized with several benchmark problems from the CPS community. The performance of the tool in comparison to state-of-the-art tools Breach [5] and DP-TaLiRo [7] is presented.

2 Overview and Features

TLTk is an object-oriented toolbox developed in `python3` (front-end) and C (back-end). An overview of the tool is presented in Figure 1. The toolbox has the following core modules:

1) The **Stochastic Optimizer** module is developed in python and is utilized to generate candidate initial conditions and input signals for the system [1]. Our implementation utilizes global optimization algorithms provided by the SciPy library[5] such as Dual Annealing [22], as well as local optimization algorithms such as Nelder-Mead [17] for refinement. In addition, due to the modular architecture of the tool, the user may develop their own or utilize any other optimization libraries in python to conduct the search. For higher dimensional predicates, the back-end robustness calculation module additionally calls the quadprog solver[6].

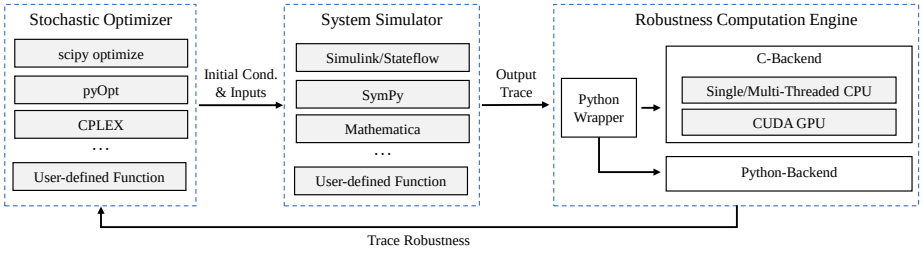

Fig. 1. An overview of TLTk and its major components.

2) The **System Simulator** module may be a standalone function, an interface for Matlab Simulink, or other simulators that support python-integration.

[5] SciPy Optimize: https://docs.scipy.org/doc/scipy/reference/optimize.html
[6] QuadProg: https://github.com/rmcgibbo/quadprog

3) The **Robustness Computation Engine** module utilizes our C back-end implementation for fast robustness computation. The module utilizes multi-threaded CPU/GPU processing to compute the robustness of a trace with respect to a specification in parallel. A python implementation is also available. Although much slower than the C implementation, the python implementation is developed so that modifications to the robustness algorithm can be made easily for utilization in application-specific case studies and even topics such as planning. For example, in [16,11], the authors consider a smooth cumulative robustness, which modifies the semantics of temporal operators to include smoothing functions in order to make it more suitable for planning problems. To setup and run the tool, several options are available:

- *Building from Source* (Linux). This option provides the best performance. However, it is restricted to the Linux OS since we are using OpenMP[7] for parallel computation. The user needs to follow a set of commands provided in the User Guide to install software dependencies and compile the programs.
- *Running through Docker* (Linux, Windows, Mac). Docker enables a single command setup for the tool and all the required dependencies. Currently, GPU functionality is accessible only on Linux hosts[8]. The TLTK docker image can be pulled using the following command: `docker pull bardhh/tltk`.
- *Python API* (Linux). In addition to the previous methods, TLTK is available as a python package and can be installed through the `pip3` Python package installer using the following command: `pip3 install tltk_mtl`. Once the package is installed, it can be imported in any `python3` script and used for robustness computation.

3 Robustness Computation

The parallel robustness computation engine builds on the dynamic programming algorithm developed in [23]. A tabular representation of subformulas of the specification (rows) and the trace samples (columns) of the signal is utilized. In the rows that contain only atomic predicates, quadratic programming is utilized to populate the cells since robustness requires computations between a point and a set [8]. Following the semantics of the robustness of temporal logic operators, the table can be dynamically collapsed to return a single robustness value for the entire trace.

Consider the specification $\phi = \Box\neg(Ax \leq b)$, which states that region $r1$ should not be reached. In Fig. 2 (a), an example trace is presented. As illustrated in the figure, the robustness value of the trace with respect to the specification should return the minimum distance ρ^* between the sample point μ^* and the unsafe set $r1$. To compute this, the distance of each sample $\mu_0, \mu_1, ..., \mu_i, ..., \mu_n$ to the

[7] OpenMP: https://www.openmp.org/

[8] Nvidia has announced that support for this functionality in Windows OS is under development. https://devblogs.nvidia.com/announcing-cuda-on-windows-subsystem-for-linux-2/

unsafe set is computed (see Fig. 2 (b)). For each sample, a quadratic program is solved to return the distance to the unsafe set.

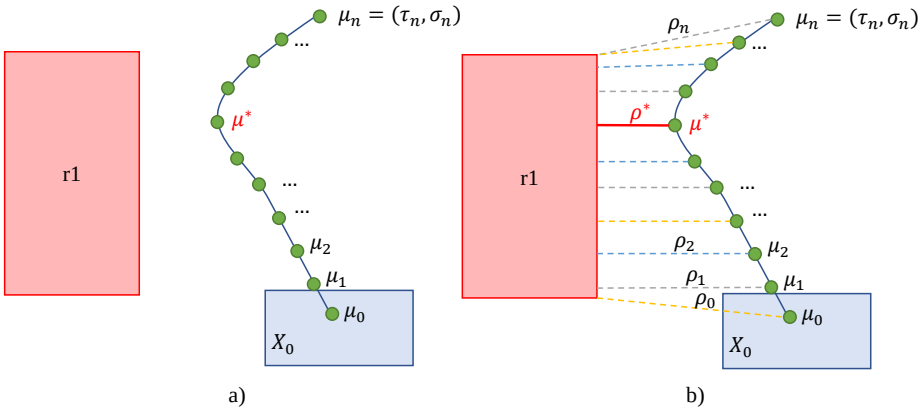

Fig. 2. Parallel Robustness Computation using TLTk.

The worst-case time complexity of the robustness computation algorithm is $O(|\varphi||\tau|c)$, where $|\varphi|$ is the length of the formula, $|\tau|$ is the number of samples, and c is the number of samples in the longest interval indicated by the timing constraints. Even though this is the same as in the algorithm presented in [23], the changes presented in this paper significantly improve the running-time of the algorithm. In addition, performance is improved due to parallel computation and the tool can handle larger traces due to more efficient memory management.

Parallel Computation: (i) Given a discrete output trace of the system composed of n time-value pairs, the robustness computation requires that for each predicate in the formula, robustness should be computed n times. This process is parallelized in TLTk with multi-threaded CPU/GPU support. The dynamic programming algorithm generates a table with entries for each subformula and predicate. The bottom rows of the table are reserved for predicates and each distance to the unsafe set is computed in parallel. (ii) Once this is completed, the tree of subformulas is traversed in parallel to return the robustness.

Average Running-time: For time bounded formulas, two improvements are made. (i) A modified binary search algorithm is developed that determines the indices that correspond to the time bounds. For every iteration of the algorithm, the indices are approximated based on the previous indices and then verified. In addition, as the formula is evaluated, the time bound under consideration is restricted to the remaining trace. (ii) For time bounded formulas, table computations flow from right to left, and there is a sliding window reflecting the time bound of the formula. Since robustness computation is an iterative series of min/max operations and most of the data in the current sliding window overlap with the previous window, we only need to consider the change between the sliding windows to calculate the min/max of the current window.

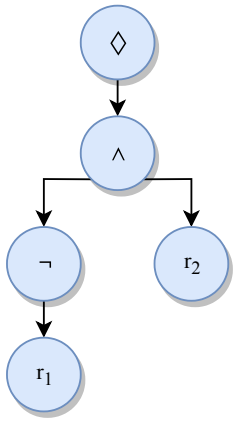

Fig. 3. Tree representation for formula $\phi = \Diamond(\neg r_1 \wedge r_2)$.

Memory Management: By dynamically allocating and deallocating memory based on the structure of the formula, TLTK operates with a significantly smaller memory footprint. This enables robustness computation of very large system traces. Initially, a formula φ is decomposed into a tree structure. For example, for $\varphi = \Diamond(\neg r_1 \wedge r_2)$ the computation flow reads from left to right, as shown in Figure 3. First, memory is allocated for predicates r_1 and r_2 and robustness is computed. Instead of allocating a new row for $\neg r_1$, an existing row where r_1 was stored is overwritten with the new values. After, the memory allocated for r_2 is utilized to store the robustness with respect to $\neg r_1 \wedge r_2$. The only additional row added is for the final expression of the eventually operator.

The robustness computation of the Always ($\Box$), Eventually ($\Diamond$), And ($\wedge$), Or ($\vee$), Not ($\neg$) and Until(U) operators is done in parallel in the C back-end. The main program and interface of TLTK is implemented in Python. A wrapping function is utilized to ensure the execution of the C code from the Python interface. Algorithms 8 and 7 are used for robustness computation.

4 Evaluation and Experimental Results

In the following, we evaluate the robustness computation times on various trace sizes with temporal logic specifications and compare to well-known tools DP-TaLiRo [23] and Breach [5]. The experiments were conducted on a desktop computer with the following specifications: CPU i7-8700K CPU @ 3.70GHz, GPU GTX 1080Ti, 32GiB DDR4 RAM and Ubuntu 18.04.3 LTS.

Comparison with Breach[9]. We present the experimental comparison of TLTK with Breach in Table 1. We compare the robustness computation of STL formulas with trace sizes ranging from 2^{10} to 2^{29}. The traces are checked against three STL specifications. A performance improvement of at least one order of magnitude can be observed.

[9] Breach 1.7 downloaded on 01.16.2020 from https://github.com/decyphir/breach

	φ_{b1}		φ_{b2}		φ_{b3}	
2^x	TLTk	Breach	TLTk	Breach	TLTk	Breach
10	0.00007	0.00350	0.00009	0.03343	0.00010	0.03588
14	0.00005	0.00935	0.00007	0.00480	0.00022	0.01243
18	0.00057	0.02322	0.00083	0.02862	0.00223	0.10138
22	0.00683	0.30040	0.01305	0.47182	0.03549	1.69170
26	0.10719	5.30410	0.21444	8.58840	0.56538	30.24000
27	0.21375	10.67300	0.42656	17.31700	1.12633	60.25600
28	0.42930	186.82000	0.85081	107.89000	2.24889	×
29	0.85353	×	1.69901	×	4.49088	×

Specification	Predicates
$\phi_{b1} = \neg(\Diamond s_1)$ $\phi_{b2} = \neg(\Diamond_{[0,1000]} s_1 \wedge \Box_{[100,300]} r_1)$ $\phi_{b3} = \neg(\Diamond_{[0,1000]} s_1 \wedge \Box_{[0,200]}(r_1 \wedge \Box(\Diamond(s_1 \wedge (s_1 U r_1)))$	$s_1 : speed(t) > 160$ $r_1 : rpm(t) < 4500$

Table 1. Comparison of computation times in seconds for TLTk and Breach with various specifications and trajectory sizes. × indicates out of memory error instances.

Comparison with DP-TaLiRo[10]. The experimental comparison of TLTk with DP-TaLiRo is presented in Table 2. The comparison is conducted using formulas that are defined for several benchmark problems. Specifically, requirements for the aircraftODE [20] (ϕ_{s1}), Navigation [20] (ϕ_{s2}), Heat Transfer [10] (ϕ_{s6}) and Nonlinear [1] (ϕ_{s3-s5}) Systems. The specifications are defined in Table 3. A performance improvement of at least two orders of magnitude can be observed.

5 Related Works

TLTk was inspired by the Matlab toolboxes S-TaLiRo [3,7] and Breach [5]. All three tools provide an automated test-case generation process for finding system behaviors that falsify temporal logic specifications. In addition to falsification, these tools provide methods for requirement mining, conformance testing and real-time monitoring. They provide various optimization algorithms for black-box and grey-box testing. A different approach to falsification is utilized in the tool FalStar [24]. In FalStar, the falsifying system behavior is generated by constructing the input signal incrementally in time. This is particularly useful for reactive specifications. In another tool, **falsify** [2], the program solves the falsification problem through reinforcement learning. The tool attempts to find falsification by observing the output signal and modifying the input signal during system simulation. A trajectory splicing/multiple shooting approach is utilized in the tool S3CAM [25], which explores the state-space of CPS and

[10] DP-TaLiRo is part of S-TaLiRo toolbox version 1.6. The tool was downloaded on 01.16.2020 from https://app.assembla.com/spaces/s-taliro_public/

	φ_{s1}		φ_{s2}		φ_{s3}	
2^x	TLTK	DP-TALIRO	TLTK	DP-TALIRO	TLTK	DP-TALIRO
10	0.0023	1.5819	0.0028	1.5995	0.0018	1.8497
12	0.0087	6.3102	0.0106	6.3334	0.0081	7.0429
14	0.0295	25.1800	0.0361	25.3340	0.0252	28.1650
16	0.1118	100.6800	0.1375	101.3300	0.1002	112.6400
18	0.4429	403.1900	0.5334	405.1800	0.4013	450.4800
20	1.7296	1610.3000	2.1250	1621.0000	1.6054	1802.5000
21	3.5078	3222.3000	4.2977	3240.0000	3.2694	3604.3000
22	7.0906	×	8.5688	×	6.4353	×
23	14.0333	×	17.1810	×	13.0045	×
24	28.0057	×	34.2625	×	26.3092	×
	φ_{s4}		φ_{s5}		φ_{s6}	
2^x	TLTK	DP-TALIRO	TLTK	DP-TALIRO	TLTK	DP-TALIRO
10	0.0018	4.3029	0.0022	6.8373	0.0072	5.6698
12	0.0081	17.2180	0.0082	27.4090	0.0211	3.1871
14	0.0257	68.8750	0.0256	109.5900	0.0843	12.7620
16	0.1012	275.4100	0.1021	438.5200	0.3409	51.0760
18	0.4030	1102.0000	0.4097	1753.2000	1.3448	204.0600
20	1.6301	4402.8000	1.6199	7014.8000	5.4648	816.7000
21	3.2396	8805.1000	3.2225	14023.0000	10.9827	1632.2000
22	6.4321	×	6.4926	×	21.8416	×
23	12.8838	×	13.1343	×	43.2679	×
24	25.6910	×	26.0209	×	87.0204	×

Table 2. Comparison of computation times in seconds for TLTK and DP-TALIRO with various trajectory sizes. Specifications φ_{s1} through φ_{s6} and the predicate definitions can be found in Table 3. Symbol × indicates out of memory instances.

splices trace segments to find a path from one state to another. This approach was later extended to incorporate symbolic reachability techniques in the tool XSPEED [4]. To enable monitoring of temporal logic specifications in robotic systems, in [19], the authors present RTAMT, which offers integration with ROS and supports monitoring of past and future time specifications.

5.1 Conclusion and Future Work

We have presented TLTK, a tool for falsification and parallel robustness computation of STL specifications. The modular architecture of the tool enables integration with any stochastic optimization algorithm or system simulator available in Python. The experimental results demonstrate that the multi-threaded CPU/GPU robustness engine shows an runtime improvement of at least one order of magnitude in comparison BREACH and DP-TALIRO.

MTL Specifications and Predicates	
$\phi_{s1} = \neg(\Box_{[5,150]} r_3 \wedge \Diamond_{[300,400]} r_4)$ $r_3 : A_{s1} * x \leq [250\ -240]^T$, $r_4 : A_{s1} * x \leq [240\ -230]^T$	$A_{s1} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}$
$\phi_{s2} = (\neg p_{11}) U p_{12}$ where $p_{11} : A_{s2} * x \leq [3.8\ -3.2\ 0.8\ -0.2]^T$, $p_{12} : A_{s2} * x \leq [3.8\ -3.2\ 1.8\ -1.2]^T$	$A_{s2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix}$
$\phi_{s3} = \Box(r_7 \wedge \Diamond_{[0,100]} r_8)$ $\phi_{s4} = \neg(\Diamond r_7 \wedge \Box(r_8 \wedge \Box(\Diamond(r_7 \wedge (r_7 U r_8)))))$ $\phi_{s5} = \neg(\Diamond r_7 \wedge \Box(r_8 \wedge \Box(\Diamond(r_7 \wedge (r_7 U r_8))))) \wedge$ $\Diamond(\Box(r_7 \vee (r_8 U r_7))))$ $r_7 : A_{s345} * x \leq [1.6\ -1.4\ 1.1\ -0.9]^T$, $r_8 : A_{s345} * x \leq [1.5\ -1.2\ 1.0\ -1.0]^T$	$A_{s345} = \begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix}$
$\phi_{s6} = \Box p$ $p : A_{s6} * x \leq$ $[14.5\ 14.5\ 13.5\ 14\ 13\ 14\ 14\ 13\ 13.5\ 14]^T$	$A_{s6} = I(10)$

Table 3. Specifications and predicates for the Signal Temporal Logic specifications utilized for the comparison between TLTk and S-TaLiRo.

As part of future work, the GPU algorithm may be improved further. In the current implementation, GPU computations are called for each predicate and temporal operator in the formula. This process causes an overhead when transferring the system trace to the GPU memory for each call. In addition, we will add the requirement mining functionality as well as integration with ROS. The goal is to use the tool for planning and control of robotic systems.

References

1. Abbas, H., Fainekos, G.E., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Probabilistic temporal logic falsification of cyber-physical systems. ACM Transactions on Embedded Computing Systems **12**(s2) (May 2013)
2. Akazaki, T., Liu, S., Yamagata, Y., Duan, Y., Hao, J.: Falsification of cyber-physical systems using deep reinforcement learning. In: Formal Methods. Lecture Notes in Computer Science, vol. 10951, pp. 456–465. Springer (2018)
3. Annapureddy, Y.S.R., Liu, C., Fainekos, G.E., Sankaranarayanan, S.: S-taliro: A tool for temporal logic falsification for hybrid systems. In: Tools and algorithms for the construction and analysis of systems. LNCS, vol. 6605, pp. 254–257 (2011)
4. Bogomolov, S., Frehse, G., Gurung, A., Li, D., Martius, G., Ray, R.: Falsification of hybrid systems using symbolic reachability and trajectory splicing. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. pp. 1–10 (2019)

5. Donze, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Computer Aided Verification, LNCS, vol. 6174, pp. 167–170 (2010)
6. Donze, A., Ferrre, T., Maler, O.: Efficient robust monitoring for stl. In: Proceedings of the 25th International Conference on Computer Aided Verification. pp. 264–279. CAV, Springer-Verlag, Berlin, Heidelberg (2013)
7. Fainekos, G., Hoxha, B., Sankaranarayanan, S.: Robustness of specifications and its applications to falsification, parameter mining, and runtime monitoring with s-taliro. In: International Conference on Runtime Verification. pp. 27–47. Springer (2019)
8. Fainekos, G.E., Girard, A., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for dynamic robots. Automatica **45**(2), 343–352 (2009)
9. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoretical Computer Science **410**(42), 4262–4291 (2009)
10. Fehnker, A., Ivancic, F.: Benchmarks for hybrid systems verification. In: Hybrid Systems: Computation and Control. LNCS, vol. 2993, pp. 326–341. Springer (2004)
11. Haghighi, I., Mehdipour, N., Bartocci, E., Belta, C.: Control from signal temporal logic specifications with smooth cumulative quantitative semantics. arXiv preprint arXiv:1904.11611 (2019)
12. Hoxha, B., Abbas, H., Fainekos, G.: Benchmarks for temporal logic requirements for automotive systems. In: Workshop on Applied Verification for Continuous and Hybrid Systems (2014)
13. Hoxha, B., Dokhanchi, A., Fainekos, G.: Mining parametric temporal logic properties in model based design for cyber-physical systems. International Journal on Software Tools for Technology Transfer **20**, 79–93 (2018)
14. Jin, X., Donzé, A., Deshmukh, J.V., Seshia, S.A.: Mining requirements from closed-loop control models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **34**(11), 1704–1717 (2015)
15. Kapinski, J., Deshmukh, J.V., Jin, X., Ito, H., Butts, K.: Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. IEEE Control Systems **36**(6), 45–64 (2016)
16. Leung, K., Aréchiga, N., Pavone, M.: Backpropagation for parametric stl. In: 2019 IEEE Intelligent Vehicles Symposium (IV). pp. 185–192. IEEE (2019)
17. Nelder, J.A., Mead, R.: A simplex method for function minimization. The computer journal **7**(4), 308–313 (1965)
18. Nghiem, T., Sankaranarayanan, S., Fainekos, G.E., Ivancic, F., Gupta, A., Pappas, G.J.: Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In: Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control. pp. 211–220. ACM Press (2010)
19. Nickovic, D., Yamaguchi, T.: Rtamt: Online robustness monitors from stl. arXiv preprint arXiv:2005.11827 (2020)
20. Sankaranarayanan, S., Fainekos, G.: Falsification of temporal properties of hybrid systems using the cross-entropy method. In: ACM International Conference on Hybrid Systems: Computation and Control (2012)
21. Tuncali, C.E., Hoxha, B., Ding, G., Fainekos, G., Sankaranarayanan, S.: Experience report: Application of falsification methods on the uxas system. In: NASA Formal Methods Symposium. pp. 452–459. Springer (2018)
22. Xiang, Y., Sun, D., Fan, W., Gong, X.: Generalized simulated annealing algorithm and its application to the thomson model. Physics Letters A **233**(3), 216–220 (1997)

23. Yang, H.: Dynamic Programming algorithm for Computing Temporal Logic Robustness. Master's thesis, Arizona State University (2013)
24. Zhang, Z., Ernst, G., Sedwards, S., Arcaini, P., Hasuo, I.: Two-layered falsification of hybrid systems guided by monte carlo tree search. IEEE Trans. on CAD of Integrated Circuits and Systems **37**(11), 2894–2905 (2018)
25. Zutshi, A., Deshmukh, J.V., Sankaranarayanan, S., Kapinski, J.: Multiple shooting, cegar-based falsification for hybrid systems. In: Proceedings of the 14th International Conference on Embedded Software. pp. 1–10 (2014)

A Robustness Computation Algorithms

Algorithm 1: NOT sub task.

```
1 Not (¬) subtask
  Data: Robustness array not_robustness
2 for i ← |τ| to 0 do
3     not_robustness[i] ← −1 ∗ not_robustness[i]
```

Algorithm 2: AND sub task.

```
1 And (∧) subtask
  Data: Robustness array left_robustness
  Data: Robustness array right_robustness
2 for i ← |τ| to 0 do
3     left_robustness[i] ← min(left_robustness[i], right_robustness)
```

Algorithm 3: OR sub task.

```
1 Or (∨) subtask
  Data: Robustness array left_robustness
  Data: Robustness array right_robustness
2 for i ← |τ| to 0 do
3     left_robustness[i] ← max(left_robustness[i], right_robustness)
```

Algorithm 4: find_max(*robustness*, *start*, *end*)

```
  Input: Robustness array robustness
  Input: start number start
  Input: end number end
  Output: maximum index max_index
1 max ← robustness[0]
2 max_index ← 0
3 for i ← start to end do
4     if max < robustness[i] then
5         max ← robustness[i]
6         max_index ← i
7     return max_index
8 ]
```

Algorithm 5: find_min($robustness, start, end$)

Input: Robustness array $robustness$
Input: start number $start$
Input: end number end
Output: minimum index min_index

```
min ← robustness[0]
min_index ← 0
for i ← start to end do
    if robustness[i] < min then
        min ← robustness[i]
        min_index ← i
    end
    return min_index
end
]
```

Algorithm 6: search_sorted($\tau, start, end, time$)

Input: time array τ
Input: start index $start$
Input: end index end
Input: Target time $time$
Output: Middle index $middle$

```
lower_index ← start
upper_index ← end − 1
middle ← (lower_index + upper_index)/2
while lower_index ≤ upper_index do
    if τ[middle] < time then
        lower_index ← middle + 1
    else if τ[middle] == time) then
        break
    else
        upper_index ← middle − 1
    end
    middle = (lower_index + upper_index)/2
end
```

Algorithm 7: FINALLY(I,ε,τ,ς)

Input: Time bound array of two numbers I
Input: Robustness array ε
Input: Time Step array τ
Input: Result robustness array ς
`/* If time bounds are 0 and ∞ perform cheaper calculation */`
1 **if** $I = [0, \infty]$ **then**
2 $max \leftarrow \varepsilon[|\tau|]$
3 **for** $i \leftarrow |\tau|$ **to** 0 **do**
4 **if** $max < \varepsilon[i]$ **then**
5 $max \leftarrow \varepsilon[i]$
6 $\varsigma[i] \leftarrow max$
7 **else**
8 $max_i \leftarrow -1$
9 $max \leftarrow -\infty$
10 **for** $i \leftarrow |\tau|$ **to** 0 **do**
11 $time_l \leftarrow \tau[i] + I[0]$
12 $time_u \leftarrow \tau[i] + I[1]$
13 $index_u \leftarrow Search\_sorted(\tau, i, |\tau|, time_u)$
`/* Checks if search needs to be performed */`
14 **if** $I[0] == 0$ **then**
15 $index_l \leftarrow i$
16 **else**
17 $index_l \leftarrow Search\_sorted(\tau, i, |\tau|, time_l)$
`/* Does full search on first iteration */`
18 **if** $max_i == -1$ **then**
19 $max_i \leftarrow Find\_max(\varepsilon, index_l, index_u)$
20 $\varsigma[i] \leftarrow \varepsilon[max_i]$
`/* check if the max has moved out of frame */`
21 **else if** $index_u < max_i$ **then**
22 $max_i \leftarrow Find\_max(\varepsilon, index_l, index_u)$
23 $\varsigma[i] \leftarrow \varepsilon[max_i]$
`/* runs if max has not moved out of frame. Is cheaper than running full search */`
24 **else**
25 $possible\_max_i \leftarrow Find\_max(\varepsilon, index_l, prev\_index_l)$
26 **if** $max \leq \varepsilon[possible\_max_i]$ **then**
27 $max_i \leftarrow possible\_max_i$
28 $\varsigma[i] \leftarrow \varepsilon[max\_index]$
29 $prev\_index_l \leftarrow index_l$
30 $max \leftarrow \varepsilon[max_i]$

Algorithm 8: GLOBAL(I,ε,τ,ς)

Input: Time bound array of two numbers I
Input: Robustness array ε
Input: Time Step array τ
Input: Result robustness array ς
`/* If time bounds are 0 and ∞ perform cheaper calculation */`
1 **if** $I = [0, \infty]$ **then**
2 $min \leftarrow \varepsilon[|\tau|]$
3 **for** $i \leftarrow |\tau|$ **to** 0 **do**
4 **if** $\varepsilon[i] < min$ **then**
5 $min \leftarrow \varepsilon[i]$
6 $\varsigma[i] \leftarrow min$
7 **else**
8 $min_i \leftarrow -1$
9 $min \leftarrow -\infty$
10 **for** $i \leftarrow |\tau|$ **to** 0 **do**
11 $time_l \leftarrow \tau[i] + I[0]$
12 $time_u \leftarrow \tau[i] + I[1]$
13 $index_u \leftarrow Search\_sorted(\tau, i, |\tau|, time_u)$
`/* Checks if search needs to be performed */`
14 **if** $I[0] == 0$ **then**
15 $index_l \leftarrow i$
16 **else**
17 $index_l \leftarrow Search\_sorted(\tau, i, |\tau|, time_l)$
`/* Does full search on first iteration */`
18 **if** $min_i == -1$ **then**
19 $min_i \leftarrow Find\_min(\varepsilon, index_l, index_u)$
20 $\varsigma[i] \leftarrow \varepsilon[max_i]$
`/* check if the max has moved out of frame */`
21 **else if** $index_u < max_i$ **then**
22 $min_i \leftarrow Find\_min(\varepsilon, index_l, index_u)$
23 $\varsigma[i] \leftarrow \varepsilon[max_i]$
`/* runs if max has not moved out of frame. Is cheaper than running full search */`
24 **else**
25 $possible\_min_i \leftarrow Find\_min(\varepsilon, index_l, prev\_index_l)$
26 **if** $min \leq \varepsilon[possible\_min_i]$ **then**
27 $min_i \leftarrow possible\_min_i$
28 $\varsigma[i] \leftarrow \varepsilon[min\_index]$
29 $prev\_index_l \leftarrow index_l$
30 $min \leftarrow \varepsilon[min_i]$