PSY-TaLiRo: A Python Toolbox for Search-Based Test Generation for Cyber-Physical Systems

Quinn Thibeault(✉), Jacob Anderson, Aniruddh Chandratre, Giulia Pedrielli, and Georgios Fainekos

Arizona State University, Tempe, AZ 85281, USA
qthibeau@asu.edu

Abstract. In this paper, we present the Python package PSY-TaLiRo which is a toolbox for temporal logic robustness guided falsification of Cyber-Physical Systems (CPS). PSY-TaLiRo is a completely modular toolbox supporting multiple temporal logic offline monitors as well as optimization engines for test case generation. Among the benefits of PSY-TaLiRo is that it supports search-based test generation for many different types of systems under test. All PSY-TaLiRo modules can be fully modified by the users to support new optimization and robustness computation engines as well as any System under Test (SUT).

Keywords: Falsification · Cyber-Physical Systems · Search-based test generation

1 Introduction

Requirements falsification for Cyber-Physical Systems (CPS) has gained prominence in recent years as a practical way to test and debug industrial complexity models and systems [18,22,24,27]. Since the automotive industry was an early adopter of the falsification technology [16], many of the benchmark CPS models driving the research were MATLAB/Simulink models [8,14,15,23]. As a result, some of the academic falsification tools are MATLAB tools: Breach [10], S-TaLiRo [5], and ARIsTEO [18]. Other academic falsification tools that participate in the ARCH falsification competition [11] are FalStar [28] (Java/Scala), zlscheck [3] (OCaml with Zelus models), and falsify [4] (ChainerRL [1] Python Library for reinforcement learning calling MATLAB functions).

However, as the autonomy and robotics research communities (and even industry) increasingly adopt Python as the preferred language for prototyping, there is a need for a falsification toolbox natively in Python. An all Python/C++ falsification framework would resolve any computational inefficiencies and compatibility issues of calling Python from MATLAB and/or vice versa. A native Python toolbox also helps to resolve incompatibilities which can be encountered

A. Lluch Lafuente and A. Mavridou (Eds.): FMICS 2021, LNCS 12863, pp. 223–231, 2021.
https://doi.org/10.1007/978-3-030-85248-1_15

when attempting to merge modules written in Python into other software ecosystems (for example, using MATLAB to call an optimizer written in Python that calls a Simulink model). The PSY-TaLiRo (or Ψ-TaLiRo) toolbox, which stands for Python SYstems' TemporAl LogIc RObustness, addresses exactly this need. It is a fully modular and extensible toolbox for temporal logic guided falsification which mirrors the S-TaLiRo [5] structure. Namely, the users can easily call different temporal logic robustness computation engines (e.g., TLTk [9], RTAMT [20]), optimizers (SciPy), and Systems under Test (SUT) while still offering a common interface and specification language syntax. PSY-TaLiRo supports multiple libraries to compute temporal logic robustness, referred to as robustness computation backends, out of the box without any additional effort. When using the RTAMT robustness computation engine, PSY-TaLiRo supports all major operating systems.

In summary, PSY-TaLiRo makes the following contributions:

1. it is an open source fully modular toolbox in Python,
2. it provides a common syntax for the temporal logic monitors, and
3. it enables testing of Software and Hardware in the loop systems.

With PSY-TaLiRo, users will be able to quickly compare different optimization and robustness computation engines without any other changes to the test setup. Currently, the PSY-TaLiRo toolbox supports only basic functionality including defining and executing models, optimizers, and specifications. Future goals for the toolbox are to support the more advanced features of S-TaLiRo, including parameter mining and time varying control points for input signal parameterization.

This toolbox is open-source and publicly available at:

https://gitlab.com/sbtg/pystaliro

Additional materials, examples, and a quick-start guide can be found on the documentation site available at:

https://sbtg.gitlab.io/pystaliro

2 Architecture

The toolbox is organized into several modules: the SUT, the specification, the optimizers, and the options (see Fig. 1). Each module defines a **Protocol** interface as defined in [17] or **Abstract Base Class (ABC)** which may be implemented or extended respectively to create specialized implementations for a particular domain. A Python protocol is used to define the expected shape of an object but implementations are not required to be sub-classes, while an ABC requires sub-classing to implement. The life-cycle of a test is started by providing a specification, a SUT, an optimizer and options object to the toolbox entry-point. Using the SUT and the specification, the toolbox generates an objective

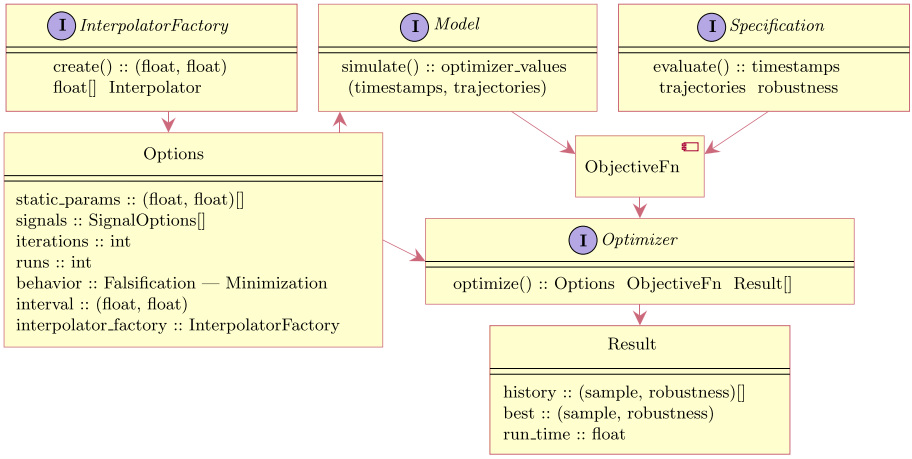

Fig. 1. Component diagram of PSY-TaLiRo architecture

function that accepts a 1-D sequence of inputs and returns a robustness value. The generated objective function and the options object are then passed as parameters to the optimizer. The optimizer executes the objective function several times, generating and storing the input sample and the output robustness for each execution. When a sample is provided to the objective function, it is decomposed into a sequence of static parameters and a sequence of signals that are used as inputs to the system model. The output of the system model is passed to the specification, which evaluates the result and produces a robustness value, which is returned to the optimizer. When the optimizer terminates its execution, a *Result* object is returned for every execution of the optimizer in case multiple experiments are performed.

Type Checking. Optional static type checking was introduced to the Python language in version 3.6 as type annotations defined in [21]. The benefit of static type checking is that multiple classes of errors can be caught before the program is executed by using a static type checker which traces the types of values through a program to ensure consistency. Python supports incremental typing, where a code-base can gradually add more type annotations over time instead of requiring the entire project to be typed immediately. PSY-TaLiRo makes extensive use of type annotations in both the internal and public APIs. Internally, annotations help ensure consistency between modules, reduce the difficulty of reasoning about functionality, and make it easier to implement additional features. For users, the annotations indicate the proper usage of the API for constructing system tests and a static type checker can provide immediate feedback.

3 Interface

The PSY-TaLiRo toolbox provides a function `staliro` which serves as an entry-point to the package. The `staliro` function accepts four required parameters - a specification, a SUT, an optimizer, an options object, and one optional parameter – an optimizer-specific options structure. Calling this function returns a sequence of `Result` objects that store the values generated by the optimizer at each iteration and the corresponding robustness value. The entry-point also implements basic validation logic for its inputs and outputs, ensuring the types of each component and their return values are correct before moving on to the next stage of the test.

3.1 System Under Test (SUT)

A SUT must provide the domain-specific information required to execute or simulate a system. It can be a simulation model e.g., (Python, MATLAB/Simulink, etc.), software-in-the-loop (SiL) (e.g., PX4, Webots, etc.), or even hardware-in-the-loop (HiL). A SUT is responsible for accepting inputs generated by the optimizer and returning the output trajectory of the execution along with the timestamps. The inputs generated by the optimizer are: static parameters, signal interpolators, and the simulation time interval. Static parameters are time-invariant inputs to the system which are often used to represent initial conditions. The simulation time interval dictates the range of time for which signals will be generated and the simulation should be executed. Signal interpolators are further described in the *Interpolation* section below.

Currently, PSY-TaLiRo provides two ways to run a SUT: a Blackbox class and an ODE integrator. The Blackbox class provides the most general way to execute a SUT because it makes no assumptions about the underlying architecture of the system it represents. To construct a Blackbox, a user needs to provide a function that accepts a vector of static parameters and/or initial conditions X, a sequence of time values T, and an array of signal values U corresponding to each time value. The Blackbox function must return the time values and corresponding output/state trajectory of the SUT. In contrast, an ODE model assumes the underlying system is represented as an ordinary differential equation and attempts to simulate the system by solving an initial-value problem. To construct an ODE model, a user must provide a function that accepts a time t, and the state at and the values of the input signal at t, and returns the derivatives of the system dynamics at time t.

Interpolation. In addition to time-invariant inputs to the SUT, PSY-TaLiRo also supports time-varying inputs referred to as signals. To generate a signal for a model, a `SignalOptions` object is created and included in the test options. A `SignalOptions` object defines an interval for the value of the signal as well as a number of **control points** which dictates how many values the optimizer should generate over the simulation interval. The optimizer-generated control points and a set of equally-spaced time values are provided to an `InterpolatorFactory`

also defined in the `SignalOptions` to create an interpolator which can generate a signal value for any time in the simulation time interval. The generated interpolators are then passed to the model under test. Currently, the PSY-TaLiRo toolbox provides factories for **PChip**, **Piecewise Linear**, and **Piecewise constant** interpolators. Should a user want to implement a custom interpolator, defining a class that implements the `InterpolatorFactory` and providing it to the `SignalOptions` object is sufficient.

3.2 Specifications

The PSY-TaLiRo toolbox supports multiple robustness computation libraries, referred to as backends by providing a uniform interface implemented as the `Specification` class. The `Specification` interface defines the `evaluate` method, which accepts the time and signal values from the SUT and returns the robustness value. It is important to note that even though PSY-TaLiRo currently supports TLTk [9] and RTAMT [20], PSY-TaLiRo's modular architecture allows the user to utilize any other robustness computation engine, or, in general, any other reward or cost function. By implementing the `Specification` interface, a user can define and use any specification language or analysis logic they choose.

To construct a specification, a user must provide a system requirement written in STL, a dictionary structure specifying the requirement data. When the TLTk library is selected, the `Specification` class is responsible for parsing the discrete time STL requirement into a corresponding TLTk object representation. ANTLRv4 is used to generate a Python parser from a discrete time Signal Temporal Logic (STL) grammar [6]. When the RTAMT library is selected, no processing is done to the requirement and the both discrete and continuous time requirements are supported.

Table 1 provides an overview of the supported common operators and syntax between the two backends. Beyond the common syntax, each robustness computation backend has different capabilities and the user is advised to read the respective documentation. For example, TLTk supports parallel computation for scaling up to very large signals and distance based robustness [12] for less conservative robustness estimates. On the other hand, RTAMT supports past-time operators and dense time semantics.

Table 1. Common TLTk [9] and RTAMT [20] syntax supported in PSY-TaLiRo.

Specification constructs	Syntax
Next*	`next`, `X`
Eventually	`eventually`, `F`
Globally	`always`, `G`
Until	`until`, `U`
Time constraints on operator OP	`OP[ ... , ... ]`
Predicates	*varName* (`<=` | `>=`) *float*

*Only supported in discrete time STL

3.3 Optimizers

An optimizer in the PSY-TaLiRo toolbox is defined as a protocol that implements a method named optimize, which accepts an objective function, an options object, and an optional object with additional configuration options that are specific to the optimizer. The optimizer is also responsible for maintaining the history of samples and robustness values generated during execution and packaging them into a `Result` object when completed. Common optimizer behavior is configured using the options object and specific optimizer behavior is configured using the optimizer-specific options object. PSY-TaLiRo also defines two search behaviors: falsification and minimization. Under falsification, the optimizer stops when the first negative robustness value is found, while minimization allows the optimizer to continue searching for lower robustness values until the execution budget is exhausted. The PSY-TaLiRo toolbox provides a Uniform Random Sampling optimizer and it also includes wrappers for Dual Annealing and Basinhopping [26] optimizers implemented in the SciPy [25] package. PSY-TaLiRo also provides support for the PartX family of optimization algorithms [7] which comes with probabilistic guarantees on the absence or presence of falsifying behaviors.

3.4 Options

To customize the behavior of the toolbox, an options object must be created and provided to the `staliro` function. Constructing a minimally valid options object can be accomplished by providing either the `static_parameters` or *signals* keyword argument to the constructor. The `static_parameters` attribute defines a sequence of intervals which represent the bounds of the input variables that do not change with respect to time. The *signals* attribute represents the opposite: a sequence of signal options objects which define system inputs that vary with time. Other important attributes are `iterations` which defines the optimizer execution budget, `runs` which specifies the number of times to execute the optimizer, and `interval` which specifies the interval of time for which the system should run.

4 Examples

PSY-TaLiRo includes as Python demo an instance of the AircraftODE benchmark [19] as well as the test setup scripts for the Python version of the F16 GCAS benchmark problem [13]. In the following, we review how PSY-TaLiRo can interface with SUT external to Python using the Blackbox template.

4.1 MATLAB/Simulink

The Simulink toolbox that is provided as a part of the MATLAB software package is useful for representing complex systems using block diagrams. MATLAB

additionally provides a Python library to enable access to the MATLAB engine from a Python application. A PSY-TaLiRo test using a Simulink model is implemented by defining a Blackbox function which uses the MATLAB Python library to pass the parameters and signal values to the Simulink simulation engine. The data returned by Simulink can then be parsed into native Python data types by the Blackbox function before returning from the `simulate` method.

There are a few considerations when implementing a Blackbox that requires the MATLAB Python library. Since the `simulate` method of the Blackbox is called many times by the optimizer, it is very inefficient to start a new instance of the MATLAB engine every time. There will also be an unavoidable time cost when interfacing with MATLAB due to the inter-process communication between the Python interpreter and the MATLAB engine. Finally, any exception that is raised during a simulation will halt the entire execution of the test, so care must be taken to ensure that any errors produced during a simulation are properly handled.

4.2 PX4

The strategies used to implement a Blackbox model that can interface with the MATLAB/Simulink engine can also be applied for communication with more complex systems such as the PX4 autopilot stack [2]. The PX4 is a commercial-grade autopilot software package used to control small aircraft like quad-rotors, and is capable of both SiL and HiL execution using one of several publicly available simulators. A successful integration of the PSY-TaLiRo toolbox and PX4 simulation environment was accomplished by using Docker to containerize the simulator and custom ground-control software to create and upload missions to the simulated drone. Some examples of requirements that were tested using the PX4 were to avoid exclusion zones when executing a mission, and another was to achieve a takeoff altitude within a threshold before landing.

5 Conclusions

We have presented the open-source Python toolbox PSY-TaLiRo (Ψ-TaLiRo). PSY-TaLiRo implements search-based test generation for falsifying temporal logic requirements over Cyber-Physical Systems (CPS). The toolbox is fully modular and extensible in order to accommodate different algorithms for optimization and temporal logic robustness (or arbitrary cost functions). Hence, PSY-TaLiRo can provide test automation support for CPS (and in particular autonomous systems) which are natively developed in Python.

Acknowledgements. This research was partially supported by DARPA (ARCOS FA8750-20-C-0507, AMP N6600120C4020) and NSF 1932068.

References

1. The ChainerRL Library. https://github.com/chainer/chainerrl
2. Open source autopilot for drones - px4 autopilot. https://px4.io
3. zlscheck: A random testing tool for Zelus. https://github.com/ismailbennani/zlscheck
4. Akazaki, T., Liu, S., Yamagata, Y., Duan, Y., Hao, J.: Falsification of cyber-physical systems using deep reinforcement learning. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 456–465. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_27
5. 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). https://doi.org/10.1007/978-3-642-19835-9_21
6. Bartocci, E., et al.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 135–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_5
7. Cao, Y., Thibeault, Q., Chandratre, A., Castillo-Effen, M., Fainekos, G., Pedrielli, G.: Work-in-progress: towards assurance case evidence generation through search based testing. In: International Conference on Embedded Software (EMSOFT) (2021, to appear)
8. Chutinan, A., Butts, K.R.: Dynamic analysis of hybrid system models for design validation. Technical report, Ford Motor Company (2002)
9. Cralley, J., Spantidi, O., Hoxha, B., Fainekos, G.: TLTk: a toolbox for parallel robustness computation of temporal logic specifications. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 404–416. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_22
10. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_17
11. Ernst, G., et al.: ARCH-COMP 2020 category report: falsification. In: 7th International Workshop on Applied Verification of Continuous and Hybrid Systems. EPiC Series in Computing, vol. 74, pp. 140–152 (2020). https://doi.org/10.29007/trr1
12. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoret. Comput. Sci. **410**(42), 4262–4291 (2009)
13. Heidlauf, P., Collins, A., Bolender, M., Bak, S.: Verification challenges in f-16 ground collision avoidance and other automated maneuvers. In: 5th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH), vol. 54, pp. 208–217 (2018)
14. Hoxha, B., Abbas, H., Fainekos, G.: Using S-TaLiRo on industrial size automotive models. In: Frehse, G., Althoff, M. (eds.) ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems. EPiC Series in Computing, vol. 34, pp. 113–119. EasyChair (2015)
15. Jin, X., Kapinski, J., Deshmukh, J.V., Ueda, K., Butts, K.: Powertrain control verification benchmark. In: 17th International Conference on Hybrid Systems: Computation and Control (2014)

16. 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 Syst. Mag. **36**(6), 45–64 (2016)
17. Levkivskyi, I., Lehtosalo, J., Langa, Ł.: Protocols: structural subtyping (static duck typing). PEP 544, Python Foundation (2017). https://www.python.org/dev/peps/pep-0544/
18. Menghi, C., Nejati, S., Briand, L.C., Parache, Y.I.: Approximation-refinement testing of compute-intensive cyber-physical models: an approach based on system identification. In: ACM/IEEE 42nd International Conference on Software Engineering (ICSE) (2020)
19. 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)
20. Nickovic, D., Yamaguchi, T.: RTAMT: Online robustness monitors from STL (2020)
21. van Rossum, G., Lehtosalo, J., Langa, Ł.: Type hints. PEP 484, Python Foundation (2014). https://www.python.org/dev/peps/pep-0484/
22. Sankaranarayanan, S., Kumar, S.A., Cameron, F., Bequette, B.W., Fainekos, G., Maahs, D.: Model-based falsification of an artificial pancreas control system. ACM SIGBED Rev. (Special Issue on Medical Cyber Physical Systems workshop (MedicalCPS 2016)) **14**(2), 24–33 (2017)
23. Strathmann, T., Oehlerking, J.: Verifying properties of an electro-mechanical braking system. In: Frehse, G., Althoff, M. (eds.) ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems. EPiC Series in Computing, vol. 34, pp. 49–56. EasyChair (2015)
24. Tuncali, C.E., Hoxha, B., Ding, G., Fainekos, G., Sankaranarayanan, S.: Experience report: application of falsification methods on the UxAS system. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 452–459. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_30
25. Virtanen, P., et al.: SciPy 1.0 contributors: SciPy 1.0: fundamental algorithms for scientific computing in Python. Nature Methods **17**, 261–272 (2020). https://doi.org/10.1038/s41592-019-0686-2
26. Wales, D.J., Doye, J.P.K.: Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms. J. Phys. Chem. A **101**(28), 5111–5116 (1997). https://doi.org/10.1021/jp970984n
27. Yamaguchi, T., Kaga, T., Donzé, A., Seshia, S.A.: Combining requirement mining, software model checking and simulation-based verification for industrial automotive systems. In: 16th Conference on Formal Methods in Computer-Aided Design (2016)
28. Zhang, Z., Ernst, G., Sedwards, S., Arcaini, P., Hasuo, I.: Two-layered falsification of hybrid systems guided by Monte Carlo tree search. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **37**(11), 2894–2905 (2018)