Open-Source Software PiTCT for Supervisory Control Design

Kai Cai and Masahiro Konishi

Abstract— **Supervisory control theory (SCT) of discrete-event systems is a mature theoretical framework with broad applications. This paper presents PiTCT, an open-source Python toolbox that provides easy and script-based access to classical SCT computation. PiTCT is designed as a Python extension of an established automaton-based control tool, exposing fundamental DES modeling, analysis, and supervisor synthesis operations through a lightweight Python interface. By embedding these operations in Python, PiTCT integrates naturally with widely used scientific and visualization packages and supports reproducible, notebook-based workflows. To further lower the entry barrier, PiTCT is distributed via PyPI and is also available through a browser-based JupyterLite service with no local installation required. The paper demonstrates the main features of PiTCT through a running printer example, illustrating how classical SCT concepts can be directly mapped to executable code.**

I. Introduction

Supervisory control theory (SCT) for discrete-event systems (DES) [1] is today supported by an ecosystem of software tools, as summarized by the IEEE CSS Technical Committee on Discrete Event Systems [2]. Classical SCT-oriented toolchains such as TCT [3], Supremica [4], CIF [5], and libFAUDES [6] provide mature implementations for automaton-based modeling, analysis, and supervisor synthesis, while other platforms emphasize timed models, temporal logic synthesis, and symbolic abstraction-based synthesis (e.g., UPPAAL [7], TuLiP [8], SCOTS [9]). Together, these tools have played a central role in advancing DES theory and its applications.

In recent years, several DES toolchains have adopted Python as an implementation language, reflecting the growing importance of *script-centric* workflows in education and prototyping. Notable examples include DESlab [10], a Python-based environment designed to support algorithm development for DES analysis and synthesis, and MDESops [11], an open-source Python library providing automaton operations, standard SCT synthesis, and advanced functionalities such as diagnosis, opacity, and security-related design. These Python-based tools demonstrate the community's interest in leveraging Python for DES research and experimentation, and provide valuable capabilities for a range of analysis and synthesis tasks.

To complement and extend this line of Python tool development, we introduce in this paper PiTCT [12], [13]. PiTCT is an open-source Python extension of the well-established TCT software package, which has been widely used for teaching and basic research on SCT for several decades. As a result, PiTCT directly inherits the core functionalities of TCT — including automaton creation, nonblocking analysis, synchronous product, controllability checking, and supervisor synthesis — whose algorithms have been extensively tested and validated through long-term educational and research use. Rather than re-implementing SCT algorithms, PiTCT exposes this mature computational backbone through a Python-native interface. With the recent open-sourcing of TCT [14], PiTCT is likewise released as open-source software [15].

The primary feature of PiTCT is therefore the modernization of access to classical SCT computation. By embedding TCT functionality into Python, PiTCT enables not only scripting, but also seamless integration with widely used scientific and visualization packages such as NumPy, SciPy, Matplotlib, Graphviz, and interactive environments such as Jupyter. This integration supports reproducible, script-based workflows that align naturally with contemporary teaching and research practices, and moreover facilitates coupling SCT computation with modern machine learning and AI-based tools (e.g. PyTorch, TensorFlow).

To further lower the entry barrier for students and non-experts, PiTCT is distributed as open-source software via PyPI with publicly available documentation [13], [15]. In addition, a browser-based JupyterLite service [16] is provided, allowing users to experiment with SCT computations without local installation. These design choices position PiTCT as a complementary tool within the DES software ecosystem: *a well-tested, easy-accessible Python toolbox for learning and prototyping supervisory control*.

The contributions of PiTCT are summarized as follows:

- An *open-source* Python toolbox for supervisory control of DES, built as an extension of the well-established TCT software.
- A *well-tested* SCT computation framework that leverages decades of validated TCT algorithms, while enabling integration with modern Python packages for education and research.
- An *easy-accessible* deployment model supporting both local installation via PyPI and browser-based use through a pre-configured JupyterLite environment.

The rest of the paper is organized as follows. Section II introduces the architecture and usage of PiTCT. Sections III and IV present core DES operations and control-oriented computations, illustrated with a running example. Section V concludes the paper with remarks on future directions.

This work was supported by JST ASPIRE Grant no. JPMJAP2519, JSPS KAKENHI Grant nos. 21H04875 and 22KK0155.

Kai Cai is with the Department of Core Informatics, Osaka Metropolitan University, Osaka 558-8585, Japan (e-mail: cai@omu.ac.jp).

II. Software Architecture and Usage

This section describes the software architecture of PiTCT, its relationship to the underlying TCT package, and the basic usage options for accessing PiTCT in both local and browser-based environments.

A. Architecture

PiTCT is designed as a Python extension of the classical supervisory control tool TCT. Its core architecture consists of two layers: a computational backend inherited from TCT, and a lightweight Python interface layer that exposes TCT functionality in a script-friendly manner.

All core SCT computations in PiTCT are executed by the TCT backend. These functions are not re-implemented in Python; instead, PiTCT invokes the existing TCT routines and returns the results in forms that can be directly manipulated in Python. As a result, PiTCT inherits the numerical correctness, algorithmic completeness, and long-term validation history of TCT.

On top of this backend, PiTCT provides Python-native functions that support scripting, automation, and interaction with external Python packages. This interface layer enables users to write concise scripts, perform batch computations, and integrate SCT operations into larger Python-based workflows.

Additional functions are included to simplify common tasks such as automaton visualization, simulation, and data exchange with standard Python data structures. Under this architecture, PiTCT can naturally evolve as new algorithms are incorporated: newly developed functions are typically implemented directly in Python and integrated with the existing TCT-based backend.

B. Local Installation and Usage

PiTCT is distributed as open-source software via PyPI, and can be installed locally using standard Python package managers. Installation instructions for macOS, Linux, and Windows are provided in [15].

After installation, PiTCT can be imported as a regular Python package as follows:

```
import pitct #import pitct package
```

The package can then be used in scripts, interactive Python sessions, or Jupyter notebooks.

Local installation is suitable for users who wish to integrate PiTCT into existing Python projects, customize their development environment, or work offline. All computation results are stored locally, allowing full control over files, scripts, and experimental data.

C. Browser-Based Usage via JupyterLite

In addition to local installation, PiTCT is available through a browser-based JupyterLite service [16]. This service allows users to access PiTCT directly from a web browser without installing any software in the local file system.

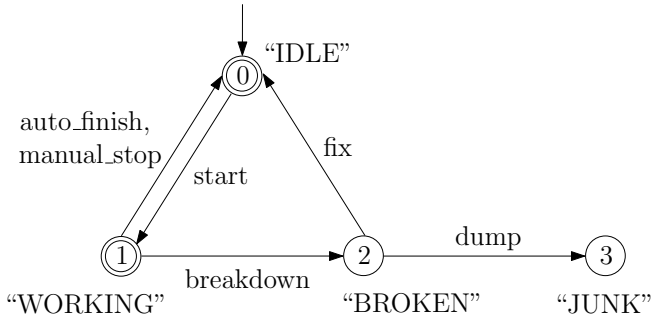

Fig. 1. Automaton model of printer

The JupyterLite environment is particularly suitable for classroom teaching, tutorials, self-study, and quick prototyping, as it eliminates local installation and configuration overhead. Users can write and execute Python code, visualize automata, and experiment with SCT computations interactively within notebook interfaces.

III. Basic Automaton Operations in PiTCT

In this section, we introduce the basic automaton operations in PiTCT, using a simple printer system as a running example. The example is intentionally kept minimal in order to emphasize fundamental automaton concepts and their executable operations.

A. Running Example: Printer

Consider a printer whose automaton model is displayed in Fig. 1. There are four *states*:

"IDLE", "WORKING", "BROKEN", "JUNK"

which are represented by circles and sequentially labeled 0, 1, 2, 3. Between the states, there are six *state transitions* (represented by directed edges):

("IDLE", start, "WORKING")
("WORKING", auto_finish, "IDLE")
("WORKING", manual_stop, "IDLE")
("WORKING", breakdown, "BROKEN")
("BROKEN", fix, "IDLE")
("BROKEN", dump, "JUNK").

Each of these transitions is written as a triple, where the first element is the state where the transition exists, the second element the event or action causing the occurrence of the transition, and the third element the state where the transition enters.

In addition, since a printer starts from the "IDLE" state, this state is the *initial state* of the automaton. In Fig. 1, the initial state 0 is with an incoming edge without source. Finally, the "IDLE" and "WORKING" states are selected as *marker states*, which are 'good states' where the printer is working normally. These two marker states are represented by double-circles.

B. Automaton Creation and Visualization

In this section, we use PITCT to create the above printer automaton, display it, and simulate its trajectories. We start with the following two lines of code, which should be included at the top of every script.

```
import pitct #import pitct package
pitct.init('xxx') #create a working folder
```

The first line imports the PiTCT package. Then a working folder "xxx" (insert any name) is created by the **init** function to save all computation results. Optionally the following allows overwriting a created folder.

```
pitct.init('xxx', overwrite=True)
#allow overwriting a created working folder
```

In PITCT, an automaton is created using the **create** function. To use this function, three elements need to be specified.

- State number: the number of states of the automaton to be created. Once a state number $n(\geq 1)$ is given, all the states are sequentially labeled from 0 to $n-1$. The initial state is labeled 0.
- Set of state stransitions: all the transition triples of the automaton. For the event labels, two formats are permitted: (i) strings like 'start', 'finish' (with single quotes); (ii) natural numbers like 0, 15 (without single quotes). Two formats are not allowed to be mixed in the same script.
- Set of marker states: a subset of the state set $\{0,\ldots,n-1\}$ that is marked.

Let's create the automaton of the printer in Fig. 1.

```
statenum=4 #number of states
#states are sequentially labeled 0,1,...,statenum
#initial state is labeled 0

trans=[(0,'start',1),
       (1,'auto_finish',0),
       (1,'manual_stop',0),
       (1,'breakdown',2),
       (2,'fix',0),
       (2,'dump',3)] # set of transitions
#each triple is (exit state, event label, entering
    state)

marker = [0,1] #set of marker states

pitct.create('PRINTER', statenum, trans, marker)
#create automaton PRINTER
```

The last line of code above uses the **create** function, which has four inputs: (i) name of the automaton to be created (a string in single quotes), (ii) state number, (iii) set of transitions, and (iv) set of marker states. This function creates an automaton PRINTER, which is saved as "PRINTER.DES" in the folder "xxx". The automaton can be visualized using the **display_automaton** function below (for this display, the package *graphviz* is used).

```
pitct.display_automaton('PRINTER')
#plot PRINTER.DES
```

The above generates the plot of the automaton PRINTER in Fig. 2. It is easily inspected that this automaton is the same as the one in Fig. 1.

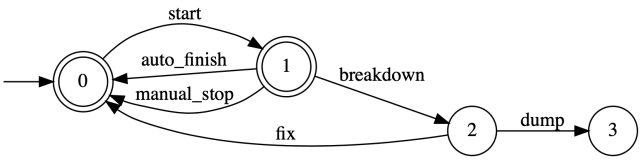

Fig. 2. PITCT plot of printer automaton

For a given automaton **A** (file "A.DES" in folder "xxx"), the functions below are handy to obtain the key information of the automaton. For the running example, we have the following.

```
pitct.statenum('PRINTER')
# Output: 4
pitct.events('PRINTER')
# Output: ['breakdown', 'dump', 'start', '
    manual_stop', 'auto_finish', 'fix']
pitct.trans('PRINTER')
# Output: [(0, 'start', 1, 'c'), (1, 'auto_finish',
     0, 'c'), (1, 'manual_stop', 0, 'c'), (1, '
    breakdown', 2, 'c'), (2, 'fix', 0, 'c'), (2, '
    dump', 3, 'c')]
pitct.marker('PRINTER')
# Output: [0, 1]
```

Note that in the output of pitct.trans('PRINTER'), each transition is attached with 'c', which indicates that all events are *controllable* by default. This point will be discussed in more detail in the next section when supervisory control is introduced.

C. Nonblocking Analysis

Let us analyze the reachability, coreachability, nonblocking, and trim properties of automaton PRINTER. First the function below checks reachability of the automaton.

```
pitct.is_reachable('PRINTER')
# Output: True
```

One can also check the reachability of an individual state. The following code checks if state 3 in PRINTER is reachable.

```
pitct.is_reachable('PRINTER',3)
# Output: True
```

In addition, if a state is reachable, a (shortest) string reaching the state from the initial state can be generated by the following function.

```
pitct.reachable_string('PRINTER', 3)
# Output: ['start', 'breakdown', 'dump']
```

The same set of the above three functions is also available for checking coreachability. First the function below checks coreachability of automaton PRINTER.

```
pitct.is_coreachable('PRINTER')
# Output: False
```

Coreachability of individual states can be checked as follows. For state 2, it is coreachable; while for state 3, it is not coreachable, and this is the reason why the automaton is not coreachable.

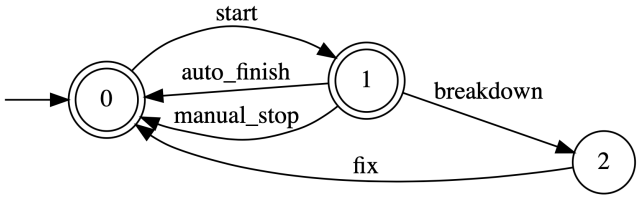

Fig. 3. PITCT trimmed automaton PRINTER_TRIM

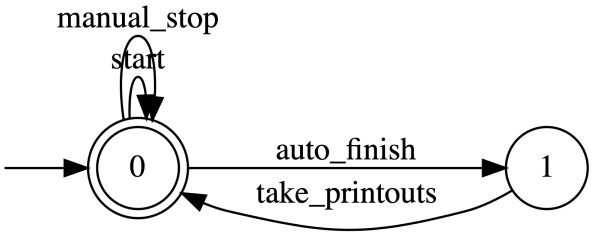

Fig. 4. PITCT plot of user automaton

```
pitct.is_coreachable('PRINTER',2)
# Output: True

pitct.is_coreachable('PRINTER',3)
# Output: False
```

Finally, for a coreachable state, a (shortest) string from the state to a marker state can be generated by the following function.

```
pitct.coreachable_string('PRINTER', 2)
# Output: ['fix']
```

It is sometimes convenient to know a shortest string from an arbitrary state (which need not be the initial state) to another (reachable) state. The following function generates a shortest string from state 1 to state 3.

```
pitct.shortest_string('PRINTER', 1, 3)
# Output: ['breakdown', 'dump']
```

Next for nonblocking of automaton PRINTER, the function below verifies that it is not, namely the automaton is blocking.

```
pitct.is_nonblocking('PRINTER')
# Output: False
```

For a blocking automaton, the following function generates the set of all blocking states (which are reachable but not coreachable).

```
pitct.blocking_states('PRINTER')
# Output: [3]
```

Finally, the following function checks if PRINTER is trim.

```
pitct.is_trim('PRINTER')
# Output: False
```

To convert a nontrim automaton to a trim automaton, the function below is used.

```
pitct.trim('PRINTER_TRIM','PRINTER')
pitct.display_automaton('PRINTER_TRIM')
```

The resulting trim automaton PRINTER_TRIM is displayed in Fig. 3. Observe that PRINTER_TRIM is the automaton resulted by removing from PRINTER the state "JUNK" and the associated transition.

D. Synchronous Product

Consider the trimmed printer automaton in Fig. 3, and another automaton USER in Fig. 4, which represents a user of this printer.

As can be seen from Fig. 4, the user can start a printing job, manually stop the job in process, and when the job finishes successfully, take the printouts. In addition, once the event auto_finish occurs (the automaton reaches state 1), user only takes printouts (i.e. event take_printouts) before s/he can start or manually stop a printing job again. This user automaton is created as follows.

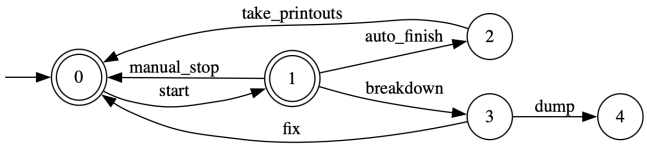

Fig. 5. PITCT synchronous product of PRINTER and USER

```
statenum=2 #number of states

trans=[(0,'start',0),
       (0,'auto_finish',1),
       (0,'manual_stop',0),
       (1,'take_printouts',0)] # set of transitions

marker = [0] #set of marker states

pitct.create('USER', statenum, trans, marker)
#create automaton USER

pitct.display_automaton('USER')
#plot USER.DES
```

Observe that these two automata share the following events in common: start, manual_stop, auto_finish. On the other hand, the events breakdown and fix belong only to PRINTER, while the event take_printouts belong only to USER. The synchronous product of these two automata is computed using the following function **sync**. The resulting automaton PU is displayed in Fig. 5.

```
pitct.sync('PU','PRINTER','USER')
pitct.display_automaton('PU')
```

Note that the state numbers of automaton PU are recoded starting from the initial state 0 and continuing sequentially. To find out which of these states corresponds to which state pair, and display the resulting PU with state pairs, the following code may be used.

```
table = pitct.sync('PU','PRINTER','USER',table=True
    , convert=True)
print(table)
# Output: 0: 0,0
#         1: 1,0
#         2: 0,1
#         3: 2,0
#         4: 3,0
pitct.display_automaton('PU')
```

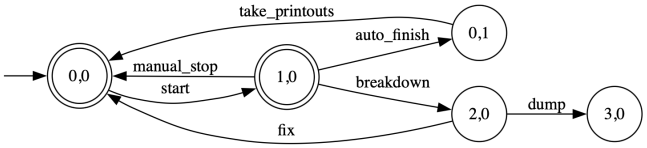

Fig. 6. PITCT sync of PRINTER and USER, displaying state pairs

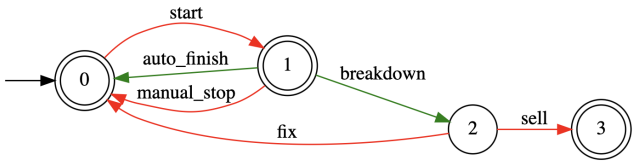

Fig. 7. PITCT create function with color coding of controllable events (red) and uncontrollable events (green)

IV. Supervisory Control Computation in PITCT

In this section, we illustrate core supervisory control computations in PITCT.

A. Controllable and Uncontrollable Events

In SCT, only *controllable* events can be disabled by a supervisor, whereas *uncontrollable* events cannot be prevented from occurring. In PITCT, adding a tag 'c' or 'u' to the transition tuple specifies the corresponding events controllable or uncontrollable, respectively (without the tag, the event is treated as controllable by default).

Now let us create a variant automaton PRINTER_SELL as the plant, with controllable and uncontrollable events specified and distinctively displayed by red and green colors, respectively. The result is in Fig. 7; the marker state 3 means "SOLD", and the "sell" event can occur at state 2 "BROKEN" to transit to state 3 "SOLD".

```
statenum=4 #number of states
#states are sequentially labeled 0,1,...,statenum-1
#initial state is labeled 0

trans=[(0,'start',1,'c'),
       (1,'auto_finish',0,'u'),
       (1,'manual_stop',0,'c'),
       (1,'breakdown',2,'u'),
       (2,'fix',0,'c'),
       (2,'sell',3,'c')] #set of transitions
#each triple is (exit state, event label, entering
    state)
#each event is either 'c' (controllable) or 'u' (
    uncontrollable)

marker = [0,1,3] #set of marker states

pitct.create('PRINTER_SELL', statenum, trans,
    marker)
#create automaton PRINTER_SELL

pitct.display_automaton('PRINTER_SELL',color=True)
#plot PRINTER_SELL.DES with color coding
```

B. Specifications and Controllability Analysis

Next we create automaton models for specifications. There are two ways. One is to use the **subautomaton** function to remove certain states and/or transitions from the plant automaton, as illustrated in the code below. The resulting automaton S1 is displayed in Fig. 8.

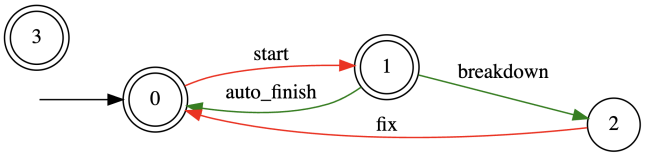

Fig. 8. Specification S1 by PITCT subautomaton function

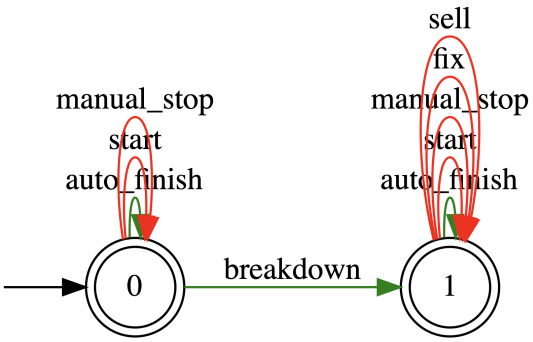

Fig. 9. Specification S2 by PITCT create function

```
pitct.subautomaton('S1','PRINTER_SELL',[3],[(1,'
    manual_stop',0)])
#create subautomaton S1 by removing from
    PRINTER_SELL [state list] and [transition list]

pitct.display_automaton('S1',color=True)
#plot S1.DES with color coding
```

The other is to use the **create** function directly. The following code creates a specification S2 which requires that the printer be allowed to breakdown only once. The automaton S2 is displayed in Fig. 9.

```
statenum=2 #number of states
#states are sequentially labeled 0,1,...,statenum
#initial state is labeled 0

trans=[(0,'start',0,'c'),
       (0,'auto_finish',0,'u'),
       (0,'manual_stop',0,'c'),
       (0,'breakdown',1,'u'),
       (1,'start',1,'c'),
       (1,'auto_finish',1,'u'),
       (1,'manual_stop',1,'c'),
       (1,'fix',1,'c'),
       (1,'sell',1,'c')] #set of transitions
#each triple is (exit state, event label, entering
    state)
#each event is either 'c' (controllable) or 'u' (
    uncontrollable)

marker = [0,1] #set of marker states

pitct.create('S2', statenum, trans, marker)
#create automaton S2

pitct.display_automaton('S2',color=True)
#plot S2.DES with color coding
```

Note that S2 is not a subautomaton of the plant, nor is the language generated by S2 a subset of that generated

by PRINTER_SELL (i.e. $L(\mathrm{S2}) \subseteq L(\mathrm{PRINTER\_SELL})$). If an automaton satisfying the above subset relation is needed, simply compute the synchronous product of S2 and PRINTER_SELL.

Controllability of the specification wrt. the plant may be checked by the **is_controllable** function (i.e. whether or not an uncontrollable transition can lead the plant to violate the specification). The following code shows how to use **is_controllable** to verify whether or not S1 is controllable wrt. PRINTER_SELL.

```
pitct.is_controllable('PRINTER_SELL','S1')
# Output: True
```

On the other hand, specification S2 turns out to be not controllable wrt. PRINTER_SELL.

```
pitct.is_controllable('PRINTER_SELL','S2')
# Output: False
```

In this case, the following **uncontrollable_states** function may be used to compute the set of all uncontrollable states of the specification (i.e. those of S2 that violate the controllability condition).

```
pitct.uncontrollable_states('PRINTER_SELL','S2')
# Output: [1]
```

C. Supervisor Synthesis

The supervisor whose behavior is the supremal controllable sublanguage of the specification language (confined by the plant language) can be computed by the function **supcon**, as illustrated in following code. First for specification S1, since it is already verified to be controllable, the result is just itself. This is confirmed by **isomorph** between C1 and the trimmed S1_trim (where the nonreachable state 3 is removed).

```
pitct.supcon('C1','PRINTER_SELL','S1')
#compute optimal supervisor

pitct.trim('S1_trim','S1')
pitct.isomorph('C1','S1_trim')
# Output: True
```

For the second specification S2 (which is not controllable), the following code computes the corresponding supervisor displayed in Fig. 10.

```
pitct.supcon('C2','PRINTER_SELL','S2')
#compute optimal supervisor

pitct.display_automaton('C2', color=True) # display
    automaton
# red transition: controllable; green transition:
    uncontrollable
```

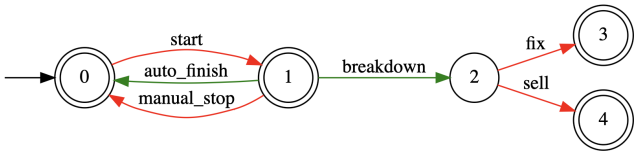

Fig. 10. Supervisor C2 by supcon

Finally we can find out the control actions of the supervisor C2 by the **conact** function.

```
table = pitct.conact('PRINTER_SELL','C2')
#compute supervisor's control actions

print(table)
# Output: 3: start
```

The above code returns that event start is disabled at state 3 of the supervisor C2, in order to prevent that the printer breaks for a second time.

V. Conclusions

This paper presented PITCT, an open-source Python toolbox for supervisory control of discrete-event systems. Built as a Python extension of the classical TCT software, PITCT inherits a mature and well-tested computational core while providing a modern, script-based interface that integrates naturally with the Python ecosystem.

Future work includes extending the Python interface with additional high-level utilities, improving visualization and interactive features, and exploring integration with data-driven and learning-based tools available in the Python ecosystem. Such extensions aim to further support education, experimentation, and interdisciplinary research, while preserving the rigorous theoretical foundations underlying supervisory control of discrete-event systems.

References

[1] W.M. Wonham and K. Cai. *Supervisory Control of Discrete-Event Systems*. Springer, 2019.

[2] IEEE CSS Technical Committee on Discrete Event Systems: Resources. https://ieeecss.org/tc/discrete-event-systems/resources, 2025. Accessed: 2025-12-31.

[3] TCT. https://www.control.utoronto.ca/~wonham/Research.html, 2025.

[4] SUPREMICA. https://github.com/Chalmers-Control-Automation-Mechatronics/Supremica/, 2022.

[5] CIF. https://eclipse.dev/escet/cif/, 2025.

[6] libFAUDES. https://fgdes.tf.fau.de/faudes/index.html, 2025.

[7] UPPAAL. http://uppaal.org, 2025.

[8] TuLiP. https://github.com/tulip-control/tulip-control, 2025.

[9] SCOTS. https://webarchiv.typo3.tum.de/static/EI/hcs/en/software/scots/, 2025.

[10] L. Carvalho L. Clavijo, J. Basilio. DESLAB: A scientific computing program for analysis and synthesis of discrete-event systems. volume 45, pages 349–355, 2012.

[11] MDESops. https://gitlab.eecs.umich.edu/M-DES-tools/desops, 2025.

[12] K. Cai. *Invitation to Supervisory Control of Discrete-Event Systems with Hands-On PyTCT*. Kindle Direct Publishing, 2024.

[13] PITCT Documentation. https://omucai.github.io/PiTCT-docs/, 2025.

[14] Open-Source TCT. https://github.com/tct-wonham, 2025.

[15] OMUCAI. PITCT (formerly PyTCT) PyPI package. https://pypi.org/project/pitct/, 2025.

[16] PITCT JupyterLite Service. https://omucai.github.io/PiTCT-docs/lite/, 2025.