PyLTA: A Verification Tool for Parameterized Distributed Algorithms

Bastien Thomas, Ocan Sankur

PyLTA: A Verification Tool for Parameterized Distributed Algorithms

Bastien Thomas and Ocan Sankur

Univ Rennes, Inria, CNRS, Rennes, France

Abstract. We present the tool PyLTA, which can model check parameterized distributed algorithms against LTL specifications. The parameters typically include the number of processes and a bound on faulty processes, and the considered algorithms are round-based and either synchronous or asynchronous.

1 Introduction

Distributed algorithms — algorithms that run on multiple communicating processes — are used in many domains including scientific computing, telecommunications and the Blockchain. Standard distributed algorithms typically perform relatively simple tasks such as consensus or leader election[18], but complexity arises from the lack of reliability of the network: some processes may crash, communications may be lost, faulty processes may send arbitrary messages (Byzantine faults)... In this setting, various automated verification techniques have been developped in order to provide guarantees on the executions of such algorithms. Notably, *parameterised* verification attempts to verify these algorithms for every possible number of processes and faults at once [4].

Threshold automata [15] (TA) are a formalism based on *counter abstraction* [19] that model asynchronous distributed algorithms with parameterised number of processes under crash and Byzantine faults. Verification can be performed using a complete encoding to SMT formulas [14]. The decidabililty of generalisations of these models was studied in [17] while [1] focuses on the complexity of the underlying problems. These algorithms were implemented in the Byzantine model checker ByMC [16]. However, algorithms based on threshold automata require bounding the diameter of the underlying transition system, either in the asynchronous case with bounded protocols (with only finitely many exchanged messages) in [15], or with unbounded messages but in the synchronous case, and for reachability properties only [21]. These techniques are therefore incomplete for threshold automata where such a bound does not exist.

In this article, we introduce PyLTA, a tool for fully verifying parameterised distributed algorithms both in the synchronous and asynchronous cases, without bounding the diameter of the state space or the number of exchanged messages. It is based on *layered threshold automata* (LTA), a formalism developped in [3] which can be thought of as some form of infinitely repeating threshold automata. These generalise the synchronous TAs used in [21] and can handle

both synchronous and asynchronous communication by exploiting some notions similar to *communication closure* [9]. This allows us to verify *any* LTL formula, including liveness properties, even on algorithms where processes may send unboundedly many messages (unlike [15] where only finite TAs and a fragment of LTL was considered).

Concretely, PyLTA takes as input the LTA description of a parameterised distributed algorithm as well as an LTL specification. It then verifies the specification under *all* parameter valuations, or finds a counterexample disproving the specification. The tool is meant to provide support for distributed algorithm designers. In fact, distributed algorithm design is not a single step process. In practice, the implemented versions of an algorithm often contain additional features or optimizations, and PyLTA can be used to automatically check these variants for counterexamples.

2 Modeling Distributed Algorithms

In order to illustrate the capabilities of PyLTA, we use the Phase King algorithm (Algorithm 1) [2]. In general, the algorithms that can be handled by PyLTA exhibit the following characteristics:

1. They are **parameterized**: in Algorithm 1, n denotes the number of processes and t a bound on the number of *Byzantine* faults. PyLTA verifies the algorithm for *all* the valuations of these parameters at once.
2. They can exchange **messages in an unbounded** domain: the indices $2i$ and $2i+1$ in Algorithm 1 are not bounded by a constant.
3. They can be **synchronous** or **asynchronous** but must ensure *communication closure*: sent and received messages are tagged with indices ($2i$ and $2i+1$ in Algorithm 1) that can only increase with time. As noted in [9], communication closure appears both in synchronous and asynchronous algorithms in the literature.
4. The algorithms should use *threshold conditions*. This means that the conditions in branches on the algorithms should be arithmetic formulas comparing *numbers* of received messages and the values of parameters (see line 10).

Under these conditions, algorithms can be encoded in an LTA. The last two conditions can often be worked around. For example, we will show along this article how Algorithm 1 can be verified despite the fact that the condition on line 6 is not ameanable to counter abstraction as it uses the identity of processes which is lost in the abstraction.

Algorithm 1 uses the parameters n, and t with the condition $t < \frac{n}{4}$. We introduce an additional paramter $f \leq t$ which is the actual number of faulty processes: the algorithm does not have access to f, but it is used during verification. Communication closure yields a layered structure of our models: a *layer* indexed by $\ell \in \mathbb{N}$ models the portion of the program that deals with messages tagged with ℓ. In Algorithm 1, the layer $\ell = 2i$ corresponds to lines 3-5, while layer $\ell = 2i + 1$ corresponds to lines 6-12.

```
1  Process PhaseKing(n, t, id, v):
       Data: n processes, t < n/4 Byzantine faults, id ∈ {0...n−1}, v ∈ {0,1}.
2      for i = 0 to t do                                   // Start of layer ℓ = 2i
3          broadcast (2i, v)                  // State a_v, v ∈ {0,1}
4          n_0 ← number of messages (2i, 0) received
5          n_1 ← number of messages (2i, 1) received// Start of layer ℓ = 2i + 1
6          if i == id then                              // Current process is king
7              if n_0 ≥ n_1 then v ← 0                                  // State k_0
8              else v ← 1                                               // State k_1
9              broadcast (2i + 1, v)
10         else if n_0 > n/2 + t then v ← 0                             // State b_0
11         else if n_1 > n/2 + t then v ← 1                             // State b_1
12         else v ← v' where (2i + 1, v') is the king's message         // State b_?
13     end
14     return v;
```

Algorithm 1: The Phase King algorithm [2] is a synchronous algorithm that solves binary consensus under $t < \frac{n}{4}$ Byzantine faults. It executes t+1 rounds, and each round $i \in \{0 \dots t\}$ is further decomposed into two layers (for round i, the layers are named $2i$ and $2i$+1). In layer $2i$, the processes broadcast their preferences v, and in layer $2i$+1, they update v either to the majority if it is strong enough, or to the preference of the process with id i, which is the king of the round i.

We use *counter abstraction* to model executions of the algorithm, meaning that we define a counter storing the number of processes at each state of the algorithm. Here, our approach differs from other works on threshold automata because we count the number of processes that *have been* through the state instead of those that are *currently* in it. It follows that the number of messages m sent during the execution can be accurately deduced from these counter values as the number of processes at states where messages m have been sent. The downside of counter abstraction is that the identities of the processes are lost. Notably, the condition on line 6 needs to be abstracted with a non deterministic choice.

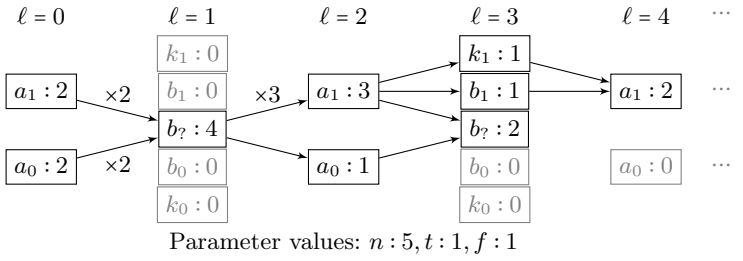


Fig. 1: A configuration of the Phase King algorithm (Algorithm 1).

Configurations. PyLTA verifies properties on all reachable *configurations*. A configuration can be interpreted as a record of events that occured during an execution. An example is depicted in Fig. 1 which we now explain.

The configuration contains an instantiation of the parameter values (given on the bottom of the figure). Moreover, for each layer index, it specifies the number of *correct* (i.e. non-faulty) processes that were at a given state at that layer; as well as the number of correct processes that moved from one state to another between consecutive layers.

In Fig. 1, initially, 2 correct processes are at state a_1, and 2 are at a_0, for a parameter valuation $n = 5, t = 1, f = 1$. Recall that layers $2i$ and $2i+1$ correspond to round i, and that the meaning of the states are given in Algorithm 1; in particular, a_x is the first line of an iteration where variable v has value x. All 4 correct processes go to $b_?$ at layer 1, which means that the Byzantine process was king at round 0. Then three of them go to a_1 at layer 3, and one of them goes to a_0, etc. This models the situation where the Byzantine process sent a message $(2 \times 0 + 1, 1)$ to the latter process but $(2 \times 0 + 1, 0)$ to the others. In the next layer, a correct process is king with value 1 (state k_1), and one correct process has received a majority of value 1 (state b_1), but not all correct processes have arrived to layer 4 yet. This configurations thus represents a finite prefix of an execution. When needed, LTL fairness assumptions can ensure that we only consider infinite configurations.

3 Input Format and Usage

The input format is based on layered threshold automata (LTA) defined in [3], which we illustrate on the running example. An input file needs to define three elements: *parameters*, *states* and *guards*.

In PyLTA, the set of parameters are declared as follows.

```
PARAMETERS: n, t, f
PARAMETER_RELATION: 4*t < n
```

The second line declares a constraint on these parameters, here $4t < n$, which is a necessary condition for the correctness of Algorithm 1.

As in our running example, the input format assumes that the states of the considered systems belong to layers. The following line defines two consecutive layers `A`, `B`, and specifies after layer `B`, we come back to layer `A` and loop.

```
LAYERS: A, B, A
```

In other terms, this results in the sequence of layers `A, B, A, B,...`. One can also specify lasso-shaped sequences; for instance, `LAYERS: A, B, B` would yield the sequence `A, B, B, B, ...`.

States can be declared by specifying the name of the layer and the name of the state separated by a period as below.

```
STATES: A.0, A.1
STATES: B.k0, B.0, B.u, B.1, B.k1
```

For instance, the first line defines the states a_0 and a_1 in Figure 1, and the second line is the rest of the states.

Transitions are defined by distinguishing cases for each state using guards. In Algorithm 1, a process needs to receive more than $\frac{n}{2} + t$ messages $(2i, 1)$ in order to move from state a_1 (line 3) to b_1 (line 11). These messages can either come from processes in state a_1 or from Byzantine processes. In PyLTA, this condition is called the *guard* from a_1 to b_1 and it is expressed with the formula $2(a_1 + f) > n + 2t$. State names correspond to the number of correct processes that have been at that state, so transitions are declared as follows.

```
FORMULA Afull: A.0 + A.1 + f == n
CASE A.1:
  IF Afull & 2*(A.1 + f) >= n THEN B.k1
  IF Afull & 2*(A.1 + f) >= n + 2*t THEN B.1
...
```

The formula `Afull` is used to enforce synchrony: no process can take a transition before every message was received. We present the other transitions for Algorithm 1 in Table 1. Note that `Afull` or an equivalent `Bfull` should also be added each time in order to avoid considering asynchronous executions.

The following instruction is used to declare an LTL specification to be verified on the configurations:

```
WITH
  A.initial: A.0 + A.1 + f == n
  A.one0: A.0 > 0
  B.not_two_kings: B.k0 + B.k1 <= 1
VERIFY: (A.initial & ! A.one0 & G(B -> B.not_two_kings)) -> G(A -> ! A.one0)
```

The instructions between `WITH` and `VERIFY` define predicates at given layers, which can be used in the subsequent LTL formula. Here, `A.one0` holds when at least one process is in state `A.0`; and `B.not_two_kings` is used to prevent executions where more than one king is present in a round. These predicates can then be used as propositions of the LTL formula that will be verified.

A layer type name (`A` or `B`) inside a formula indicates a predicate that only holds in the corresponding layers. An interpretation of the formula can therefore

Table 1: The guards of the transitions for Algorithm 1. The table on the left is for transitions leaving states of layers $\ell = 2i$, and the table on the right is for those with layer $\ell = 2i + 1$. Each cell is the guard of the transition from the state of the row to the state of the column.

$\ell = 2i$	k_0	b_0	$b_?$	b_1	k_1
a_0	$2(a_0 + f)$	$2(a_0 + f)$	$2a_0 \leq n + 2t$	$2(a_1 + f)$	$2(a_1 + f)$
a_1	$\geq n$	$\geq n + 2t$	$\wedge 2a_1 \leq n + 2t$	$\geq n + 2t$	$\geq n$

$\ell = 2i + 1$	a_0	a_1
k_0	true	false
b_0		
$b_?$	$k_1 = 0$	$k_0 = 0$
b_1	false	true
k_1		

be the following: "if there are `n` processes, and no process in `A.0`, and there is always at most one non-Byzantine king in layers of type `B`, then at all layers of type `A`, there is no process in `A.0`."

4 Tool Overview and Usage

PyLTA is written in Python. In addition to *counter abstraction* and *predicate abstraction*, PyLTA performs counter-example guided abstraction refinement [7]. Since we are working in an unbounded domain due to parameters, the tool uses an SMT solver to check the realizability of the traces, and refine the abstraction using interpolants produced by the solver [13]. The current version uses MathSAT [6] via PySMT [12]. We use Lark[20] for parsing.

The LTL specification is first negated, and then converted into a Büchi automaton using Spot [11]. The product between this automaton and the predicate abstraction is then built dynamically. We check the language emptiness of the resulting product automaton; if it is empty, then the specification holds. Otherwise, the abstract counterexample is checked for realizability using the SMT solver, and either the counterexample is confirmed, or the abstraction is refined.

We run PyLTA on an input file as follows.

```
python -m pylta [input_file]
```

The output on the file corresponding to our running example is the following:

```
VERIFYING R.initial & ! R.one0 & G (B -> B.not_two_kings) ...
Formula is Valid
```

More details such as the abstract counter examples encountered and the added predicates can be obtained by adding a `-v` flag. In this case, a single refinement was necessary, which added the predicate `B.k0 + B.0 + B.u <= 0`.

The verification algorithm does not require user interaction since abstractions are refined automatically. However, any predicate defined in the `VERIFY` instruction is used in the predicate abstraction, even if it does not appear in the formula. This behaviour provides a way to manually add predicates in order to help with the verification. The tool is distributed under the GNU GPL 3.0 licence and is available at https://gitlab.com/BastienT/pylta.

5 Conclusion

We have presented PyLTA, a tool for verifying parameterised distributed algorithms. Despite the undecidability barrier even in simple versions of the problem [21], PyLTA is able to verify complex properties on distributed algorithms, and unlike previous works, makes no assumptions on bounds on the state space or exchanged messages. As future work, one might explore the use of implicit predicate abstraction [22] to speed up the verification process. Another direction would be to integrate well ordered functions providing termination arguments [8] as used in [10] which could extend the usability of PyLTA.

References

1. A. R. Balasubramanian, Javier Esparza, and Marijana Lazić. Complexity of verification and synthesis of threshold automata. In *Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA'20)*, volume 12302 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2020.
2. Piotr Berman and Juan A. Garay. Cloture votes: n/4-resilient distributed consensus in t+1 rounds. *Mathematical Systems Theory*, 26(1):3–19, 1993.
3. Nathalie Bertrand, Bastien Thomas, and Josef Widder. Guard automata for the verification of safety and liveness of distributed algorithms. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 15:1–15:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
4. Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
5. Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. Tight bounds for k-set agreement. *J. ACM*, 47(5):912–943, 2000.
6. Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.
7. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, sep 2003.
8. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 415–426. ACM, 2006.
9. Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder. Communication-closed asynchronous protocols. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV'19)*, volume 11562 of *Lecture Notes in Computer Science*, pages 344–363. Springer, 2019.
10. Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta, and Sergio Mover. Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 271–291. Springer, 2016.
11. Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016.
12. Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT Workshop 2015*, 2015.
13. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 232–244, New York, NY, USA, 2004. Association for Computing Machinery.

14. Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, pages 719–734, 2017.
15. Igor Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Information and Computation*, 252:95–109, 2017.
16. Igor Konnov and Josef Widder. Bymc: Byzantine model checker. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III*, volume 11246 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2018.
17. Jure Kukovec, Igor Konnov, and Josef Widder. Reachability in parameterized systems: All flavors of threshold automata. In *Proceedings of the 29th International Conference on Concurrency Theory (CONCUR'18)*, volume 118 of *LIPIcs*, pages 19:1–19:17, 2018.
18. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
19. Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, page 107–122, Berlin, Heidelberg, 2002. Springer-Verlag.
20. Erez Shinan. Lark. `https://github.com/lark-parser/lark/`, 2018-2022.
21. Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. Verifying safety of synchronous fault-tolerant algorithms by bounded model checking. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19)*, volume 11428 of *Lecture Notes in Computer Science*, pages 357–374, 2019.
22. Stefano Tonetta. Abstract model checking without computing the abstraction. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2009.

A Tool Demonstration

In this section, we detail how the tool will be demonstrated. We will start by giving an example of a simple distributed algorithm, called the Flood-Min Algorithm, that can be modeled and verified with PyLTA. This algorithm is simpler than Phase King described in the core of the paper; although it does not illustrate all the capabilities of the tool, it is more convenient and easier to understand for an oral presentation.

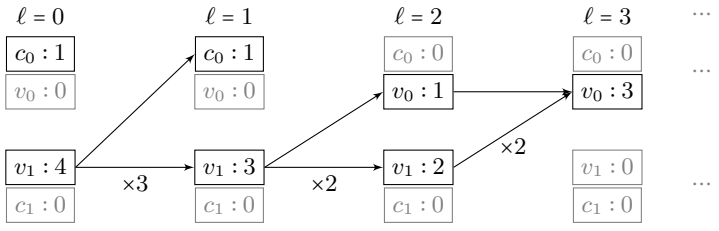
We will thus focus on the modeling aspects, and the usage of the tool by showing a property that is easy to prove, and another property that requires some interaction, thus showing the strengths but also the limitations of the tool.

The Flood-Min algorithm is from [5] and given in Algorithm 2.

1 **Process** FloodMin(n, t, v)**:**
  Data: n processes, t allowed crashes, $v \in \{0, 1\}$.
2 **for** $i = 0$ *to* t **do** `// t + 1 iterations`
3 broadcast v
4 receive values $u_0, \ldots, u_{l-1}$ from all
5 $v \leftarrow \mathsf{min}\{u_0, \ldots, u_{l-1}\}$
6 **end**
7 return v;

Algorithm 2: The Flood-Min algorithm [5], is a synchronous algorithm that solves consensus under at most t crashes.

We will directly explain configurations and the semantics of threshold automata without excessively formalizing these. A configuration can be thought of as some records of events that happened up to a point of an execution. A configuration of Algorithm 2 is represented on Figure 2.



Parameter values: $n : 5, t : 2$

Fig. 2: A configuration of the Flood-Min algorithm (Algorithm 2).

Figure 2 specifies the valuation of parameters n and t. Additionally, each column contains the possible states of a process at each iteration of the loop.

Such a column is called a *layer*. The shown configuration contains the number of processes that are at each state of each layer. For example, state v_1 (resp. v_0) in layer $\ell = 0$ has value 4 (resp. 0) meaning that 4 (resp. 0) correct processes had $v = 1$ (resp. $v = 0$) at the first iteration of the loop.

The states c_0 (resp. c_1) represent processes that had values 0 (resp. 1) but have crashed during the execution of the round. It is important to represent them separately since these processes may not have sent their values to all other processes.

Indeed, in the next layer, we see that a single process updates its value to 0 (by going to state c_0), meaning that it received a value 0 while the others only received 1's. This illustrates the correction argument of the algorithm that depends on the existence of a round where no process crashes (guaranteed by executing $t + 1$ rounds).

Next, we will show how this algorithm is described in PyLTA's input format. The parameters and states of these configurations are defined as follows:

```
PARAMETERS: n, t
LAYERS: L, L
STATES: L.c0, L.v0, L.v1, L.c1
```

The first line defines the two parameters `n` and `t`. The second line says that there is only one layer `L`, and that each layer `L` is followed by another layer `L`. The last line defines the four states that appear in layers `L`.

Next, we show how to specify the transitions between the defined states. In most cases, this depends on the number of processes in other states *of the same layer*. For example, a process moves from v_1 to v_0 if it receives a message 0 in the current round. This can only happen if another process is in state v_0 or c_0. This condition is encoded with the linear arithmetic expression $v_0 + c_0 > 0$ where v_0 and c_0 are interpreted as the *number* of processes in the corresponding state. Conversely, a process moves from v_1 to v_1 (of the next layer) if it did not receive any message 0. This can be encoded with the condition $v_0 = 0$, we omit c_0 in this case because *it is possible* for the process to not receive a message from a crashing process.

In PyLTA, this results in the following code:

```
CASE L.v0:
  IF TRUE THEN L.v0
  IF TRUE THEN L.c0

CASE L.v1:
  IF L.v0 + L.c0 > 0 THEN L.v0
  IF L.v0 + L.c0 > 0 THEN L.c0
  IF L.v0 == 0 THEN L.v1
  IF L.v0 == 0 THEN L.c1
```

We do not need to give the successors of c_0 and c_1, processes in these states will simply not take part in subsequent layers.

Last, we show how to define properties to be verified by PyLTA. First, we will show how to prove the *validity* property: 'If no process has $v = 0$, then no process

will ever have $v = 0$'. In order to encode such a property, we need a predicate that states that a process has $v = 0$, this can be done with $\text{one}_0 : v_0 + c_0 > 0$. Then the property can be stated in linear temporal logic as $\neg\text{one}_0 \longrightarrow \mathbf{G}\neg\text{one}_0$. In PyLTA, this is written as follow:

```
WITH
  L.one0: L.v0 + L.c0 > 0
VERIFY: ! L.one0 -> G ! L.one0
```

By this example, we illustrate how predicates (such as `L.one0`) can be defined, as used in temporal logic properties.

On this example, PyLTA immediately outputs:

```
Formula is Valid
```

Now, let us try to prove the termination of this algorithm. This can only hold under a fairness property. In this case, an appropriate fairness property can be that at every layer there are at least $n-t$ non-crashing processes, $\text{fair} : v_0 + v_1 >= n - t$. Under this assumption, a termination condition can be decided : $v_0 + c_0 = 0$ or $v_1 + c_1 = 0$. We will also add the initial condition $\text{ini} : v_0 + c_0 + v_1 + c_1 = n$. Then, we can attempt to give PyLTA the following input:

```
WITH
  L.ini: L.v0 + L.c0 + L.v1 + L.c1 == n
  L.fair: L.v0 + L.v1 >= n - t
  L.decided: L.v0 + L.c0 == 0 | L.v1 + L.c1 == 0
VERIFY: L.ini & G L.fair -> F L.decided
```

We now run PyLTA on this model, but it fails:

```
Cannot verify formula
```

In order to understand what happened, we run PyLTA again with the verbose flag:

```
python -m pylta -v ....
```

We obtain the following output:

```
Found an abstract counter example:
L:       0 1
ini:     T F
fair:    T T
decided: F F

Loop: 1
Prefix concretisation succeeded but loop failed
Cannot verify formula
```

We see that PyLTA attempted to instantiate an abstract path (i.e. a Boolean valuation of the predicates). It managed to instantiate a finite prefix of the path, but did not manage to concretise the loop. In other terms, it was not able to determine whether for some valuation of the parameters, there exists an *infinite* execution matching the abstract counterexample lasso.

In this case, *every finite prefix* of the path is concretisable, meaning that with predicate abstraction alone, PyLTA will not be able to solve this problem. This shows a limitation of the tool which attempts to solve an undecidable problem. Future directions to handle these cases are discussed in the paper and will be shortly mentioned in the demonstration.

We will then show how to bypass this issue in this case. In fact, the reason why the algorithm is iterated $t+1$ time is to provide one *clean* round, meaning a round where no process crashes giving the predicate clean : $c_0 + c_1 = 0$. Assuming the existence of such a round gives us the following:

```
WITH
  L.ini: L.v0 + L.c0 + L.v1 + l.c1 == n
  L.fair: L.v0 + L.v1 >= n - t
  L.clean: L.c0 + L.c1 == 0
  L.decided: L.v0 + L.c0 == 0 | L.v1 + L.c1 == 0
VERIFY: L.ini & G L.fair & F L.clean -> F L.decided
```

Under this assumption, PyLTA immediately manages to verify the property.