

# opaa1: A Lattice Model Checker

Andreas Engelbrecht Dalsgaard, René Rydhof Hansen, Kenneth Yrke Jørgensen,  
Kim Gulstrand Larsen, Mads Chr. Olesen, Petur Olsen, and Jiří Srba

Department of Computer Science, Aalborg University,  
Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark.  
{andrase,rrh,kyrke,kgl,mchro,petur,srba}@cs.aau.dk

**Abstract.** We present a new open source model checker, `opaa1`, for automatic verification of models using lattice automata. Lattice automata allow the users to incorporate abstractions of a model into the model itself. This provides an efficient verification procedure, while giving the user fine-grained control of the level of abstraction by using a method similar to Counter-Example Guided Abstraction Refinement. The `opaa1` engine supports a subset of the UPPAAL timed automata language extended with lattice features. We report on the status of the first public release of `opaa1`, and demonstrate how `opaa1` can be used for efficient verification on examples from domains such as database programs, lossy communication protocols and cache analysis.

## 1 Introduction

Common to almost all applications of model checking is the notion of an underlying concrete system with a very large—or sometimes even infinite—concrete state space. In order to enable model checking of such systems, it is necessary to construct an abstract model of the concrete system, where some system features are only modelled approximately and system features that are irrelevant for given verification purposes are “abstracted away”.

The model checker `opaa1` allows for such abstractions to be specified through user-defined lattices that are part of the model. We call them lattice automata. Lattice automata are synchronising extended finite state machines which may include lattices as variable types. The lattice elements are ordered by the amount of behaviour they induce on the system, that is, larger lattice elements introduce more behaviour. We call this the *monotonicity property*.

Lattice automata, as implemented in `opaa1`, are a subclass of well-structured transition systems [1]. The tool can exploit the ordering relation to reduce the explored state space by not re-exploring a state if its behaviour is *covered* by an already explored state. In addition to the ordering relation, lattices can have a *join operator* (the least upper bound), joining two lattice elements into one, thereby potentially overapproximating the behaviour, with the gain of a reduced state space. The overapproximated model checking can however be inconclusive. We introduce the notion of a *joining strategy*, by specifying which lattice elements are joinable, allowing the user to specify the amount of overapproximation with

much more control. This allows for a form of user-controlled CEGAR (Counter-Example Guided Abstraction Refinement) [2, 3]. The CEGAR approach can easily be automated by the user, by exploiting application-specific knowledge to automatically derive more fine-grained joining strategies given a spurious error trace. Using joining strategies can thus, for some systems and properties, provide very efficient model checking and conclusive answers at the same time.

The `opaal` tool is released under an open source licence, and can be freely downloaded at our webpage: [www.opaal-modelchecker.com](http://www.opaal-modelchecker.com). The `opaal` is available both in a GUI and CLI version, shown in Fig. 1. The UPPAAL [4] GUI is used for creation of models.

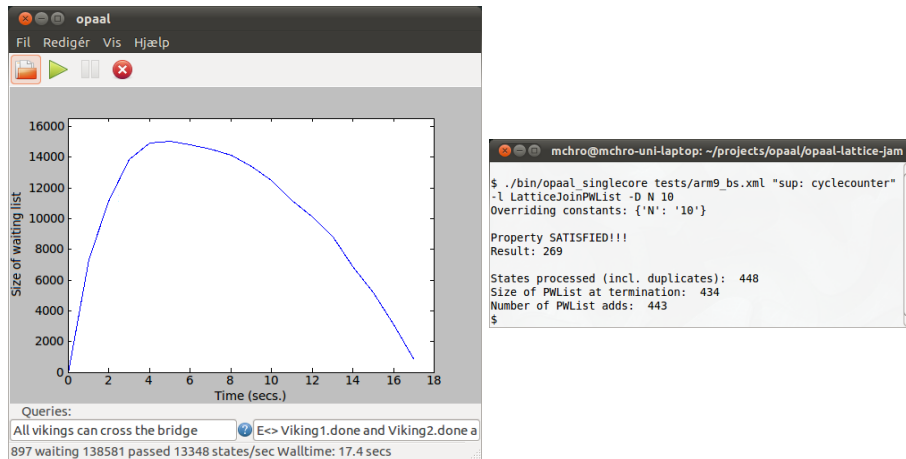


Fig. 1: (a) Screenshot of the `opaal` GUI. (b) Screenshot of the `opaal` CLI.

The `opaal` tool is implemented in Python and is a stand-alone model checking engine. Models are specified using the UPPAAL XML format, extended with some specialised lattice features. Using an interpreted language has the advantage that it is easy to develop and integrate new lattice implementations in the core model checking algorithm. Our experiments indicate that although `opaal` uses an interpreted language, it is still sufficiently fast to be useful.

Users can create a new lattice by implementing a simple Python class interface. The new class can then be used directly in the model (including all user-defined methods). Joining strategies are defined as a Python function.

An overview of the `opaal` architecture is given in Fig. 2, showing the five main components of `opaal`. The “Successor Generator” is responsible for generating a transition function for the transition system based on the semantics of UPPAAL automata. The transition function is combined with one or more lattice implementations from the “Lattice Library”, which can be easily extended with further user-defined lattices.

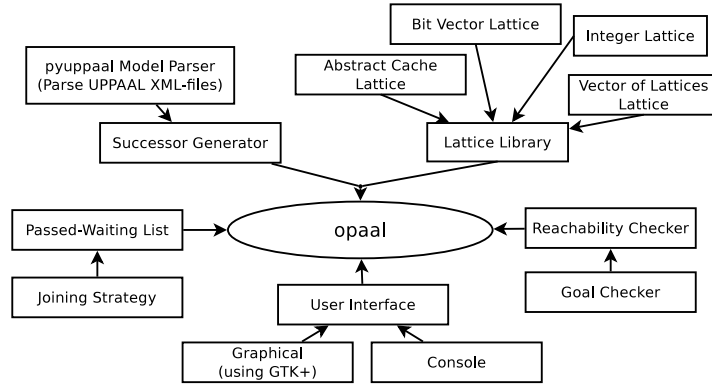


Fig. 2: Overview of opaal’s architecture.

The “Successor Generator” exposes an interface that the “Reachability Checker” can use to perform the actual verification. During this process a “Passed-Waiting List” is used to save explored and to-be explored states. The “Passed-Waiting List” uses a user-provided “Joining Strategy” on the lattice elements of states, before they are added to the list.

## 2 Examples

In this section we present a few examples to demonstrate the wide applicability of opaal. The tool has currently a number of readily available lattices which are used to abstract the real data in our examples.

### 2.1 Database Programs

In recent work by Olsen et al. [5], the authors propose using present-absent sets for the verification of database programs. The key idea is that many behavioural properties may be verified by only keeping track of a few representative data values.

This idea can be naturally described as a lattice tracking the definite present- and absent-ness of database elements. In the model, this is implemented using a bit-vector lattice. For the experiment we adopt a model from [5], where users can login, work, and logout. The model has been updated to fit within the lattice framework, as shown in Fig. 3(a). In the code in Fig. 3(b), the construct `extern` is used on line 3 to import a lattice from the library. Subsequently two lattice variables, `pLogin` and `aLogin`, are defined at line 4 and 5, both vectors of size `N_USERS`. The lattice variables are used in the transitions of the graphical model, where e.g. a special method “`num0s()`” is used to count the number of 0’s in the bitvector. The definition of a lattice type in Fig. 3(c) is just an ordinary Python class with at least two methods: `join` and the ordering.

We can verify that two users of the system cannot work at the same time using explicit exploration, or by exploiting the lattice ordering to do cover checks, see Fig. 4.

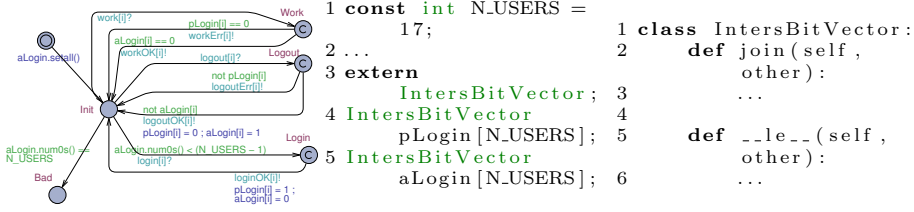


Fig. 3: (a) State machine of database model. (b) Definition of lattice variables in model code. (c) Lattice definition from lattice library, in Python.

Number of users	explicit exploration	cover check
2	224 (<1s)	56 (<1s)
3	2352 (2s)	336 (<1s)
4	21952 (28s)	1792 (2s)
5	192080 (8:22m)	8960 (9s)
6	-	43008 (48s)
7	-	200704 (4:38m)

Fig. 4: Explored states and time for the property “no two users work at the same time”.

Another property to check is that the database cannot become full. For this property we can exploit a CEGAR approach: A naïve joining strategy will give inconclusive results, but refining the joining strategy not to join two states if the resulting state has a full database, leads to conclusive results while still preserving a significant speedup, see Fig. 5.

Number of users	explicit exploration	joining (naïve strategy)	joining (refined strategy)
8	6312 (15s)	(Inconclusive) 51 (<1s)	787 (1s)
9	14228 (56s)	(Inconclusive) 57 (<1s)	1238 (2s)
10	31614 (4:19m)	(Inconclusive) 63 (<1s)	976 (2s)
11	69478 (21:35m)	(Inconclusive) 69 (<1s)	1036 (2s)
12	-	(Inconclusive) 75 (<1s)	1707 (3s)
16	-	(Inconclusive) 99 (<1s)	25900 (4:18m)
17	-	(Inconclusive) 105 (<1s)	66490 (25:01m)

Fig. 5: Explored states and time for the property “database cannot become full”.

## 2.2 Asynchronous Lossy Communication Protocol: Leader Election

Communication protocols where messages are asynchronously passed via an unreliable (lossy and duplicating) medium can be modelled as a lattice automaton.

Number of agents	explicit exploration	cover check	joining
5	840 (5s)	37 (<1s)	17 (<1s)
6	5760 (5:20m)	58 (<1s)	23 (<1s)
7	45360 (671:02m)	86 (1s)	30 (<1s)
15	-	682 (4:21m)	122 (2s)
25	-	2927 (283:16m)	327 (12s)
50	-	-	1277 (4:19m)
100	-	-	5052 (98:45m)

Fig. 6: Explored states and time for the leader election protocol.

As long as we are interested in safety properties, such a communication can be modelled as a set of already sent messages called *pool*. Initially the set *pool* is empty. Once a message is sent, it is added to the set *pool* and it remains there forever (duplication). As the protocol parties are not forced to read any message from *pool* and we ask about safety properties, lossiness is covered by the definition too.

It is obvious that  $2^{pool}$ , i.e. the set of all subsets of *pool*, together with the subset ordering is a complete lattice. As long as the set of messages is finite and all parties in the protocol behave in the way that their steps are conditioned only on the presence of a message in the pool and not on its absence, the system will satisfy the monotonicity property and we can apply our model checker.

We have modelled the asynchronous leader election protocol [6] in *opaa1*. Here we have  $N$  agents with their unique identifications  $0, 1, \dots, N - 1$  and they select a leader with the highest id. Experimental data, for the property that only the agent with the highest id can become leader, are provided in Fig. 6. The cover check column refers to using only the monotonicity property to reduce the explored state-space. We can see that while being exact (no overapproximation), the speed-up is considerable. Moreover, using the join strategy provides even more significant speed-up while still providing conclusive answers.

### 2.3 Cache Analysis

To ensure safe scheduling of real-time systems, the estimation of Worst-Case Execution Time (WCET) of each task in a given system is necessary [7]. One major part of determining WCETs for modern processors is accounting for the effects of the memory cache. Efficient abstractions exist for analysing some types of caches [8], which we have implemented as a lattice. By recasting the cache analysis into our framework we gain the ability to give WCET guarantees, and gradually refine those guarantees by being more and more concrete with respect to the data-flow of the program.

On a simple program (binary search in array of size 100) and a simple cache we get the same WCET using all approaches. The complete state space has 5726 states (computed in 6s), cover update reduces this to 4043 states (3s), while join

only needs to store 3944 states (3s). On more complex examples join will start to give overapproximated guarantees, which can be further refined.

## 2.4 Timed Automata

It is well-known that the theory of *zones* of timed automata (see e.g. [9, 10]) is a finite-state abstraction of clock values with a lattice structure. A zone-lattice is currently being developed for use in `opaal`, but has not matured to a point where meaningful experiments can be made yet.

## 3 Conclusion

We presented a new tool `opaal`, a model checker for lattice automata, and we provided a number of examples of its applicability. The expressiveness of the formalism, derived from well-structured transition systems, promises a broad applicability of the tool. Our initial experiments indicate that careful abstraction using the techniques implemented in `opaal` lead to efficient verification.

We plan on extending the foundations of `opaal` to additional formalisms such as Petri nets, as well as on improving the performance of the tool by rewriting core parts in a compiled language. Of course, additional lattices and areas of application are also to be investigated.

## References

1. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! *Theoretical Computer Science* **256**(1-2) (2001) 63–92
2. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. (2002)
3. Ball, T., Rajamani, S.: The SLAM toolkit. In: *CAV*. (2001)
4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: *Formal Methods for the Design of Real-Time Systems: SFM-RT 2004*. (2004)
5. Olsen, P., Larsen, K.G., Skou, A.: Present and absent sets: Abstraction for testing of reactive systems with databases. In: *Sixth Workshop on Model-Based Testing*, Paphos, Cyprus (March 2010)
6. Garcia-Molina, H.: Elections in a distributed computing system. *IEEE Trans. Comput.* (1982)
7. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: *The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools*. *TECS* (2008)
8. Alt, M., Ferdinand, C., Martin, F., Wilhelm, R.: Cache Behavior Prediction by Abstract Interpretation. In: *SAS 96: Proceedings of the Third International Symposium on Static Analysis*. (1995)
9. Henzinger, T., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. *Information and Computation* (1994)
10. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In Desel, J., Reisig, W., Rozenberg, G., eds.: *Lectures on Concurrency and Petri Nets*. Volume 3098 of *LNCS*. Springer Berlin / Heidelberg (2004) 87–124

## A Theory

For completeness sake we here include some preliminaries, our formal definition of lattice automata, and a formal description of the algorithms we use in opaal.

### A.1 Preliminaries

A *partial order* on a set  $L$  is a reflexive, anti-symmetric and transitive relation  $\sqsubseteq \subseteq L \times L$ . The pair  $(L, \sqsubseteq)$  is called a *partially ordered set*. Let  $(L, \sqsubseteq)$  be a partially ordered set and let  $X \subseteq L$ . An element  $\ell \in L$  is an *upper bound* of  $X$  if  $x \sqsubseteq \ell$  for every  $x \in X$ . If furthermore  $\ell \sqsubseteq \ell'$  for all upper bounds  $\ell'$  of  $X$  then  $\ell$  is the *least upper bound* of  $X$  and is denoted as  $\bigsqcup X$ . The binary least upper bound  $\bigsqcup\{x, y\}$  is written as  $x \sqcup y$ . The notion of *lower bounds* of  $X$  and the *greatest lower bound* of  $X$  (denoted by  $\bigsqcap X$  if it exists) are defined analogously. The binary greatest lower bound  $\bigsqcap\{x, y\}$  is written as  $x \sqcap y$ . Note that least upper bounds and greatest lower bounds for a given set  $X$  are unique.

An element  $\ell \in L$  such that  $\ell \sqsubseteq \ell'$  for all  $\ell' \in L$  is called the *least element* of  $(L, \sqsubseteq)$  and is denoted as  $\perp_L$  or  $\perp$  when  $L$  is clear from the context. Conversely, the *greatest element* of  $(L, \sqsubseteq)$  is an element  $\ell \in L$  such that  $\ell' \sqsubseteq \ell$  for all  $\ell' \in L$  and is denoted as  $\top_L$  or  $\top$  when  $L$  is clear from the context.

**Definition 1 (Join Semi-Lattice and Lattice).** A partially ordered set  $\mathcal{L} = (L, \sqsubseteq)$  where  $L \neq \emptyset$  is a join semi-lattice if  $\ell \sqcup \ell'$  exists for all  $\ell, \ell' \in L$ . If moreover  $\ell \sqcap \ell'$  exists for all  $\ell, \ell' \in L$  then  $\mathcal{L}$  is called a lattice.

*Example 1.* Consider the lattice for the abstraction of a database into present and absent sets mentioned earlier. Let  $D \subseteq \mathcal{U}$  be a database containing elements from some universe  $\mathcal{U}$  of possible values. As an abstraction over this universe we select a small set of representative values,  $\mathcal{S} \subseteq \mathcal{U}$ . We present two subsets of  $\mathcal{S}$  the *present set*  $P$ , and the *absent set*  $A$ . Adapting terminology from the abstract interpretation community, this forms a *must analysis*, that is, a value in the present set *must* be in the database, and a value in the absent set *may not* be in the database.

The set of databases abstracted over with a concrete set of present and absent sets is the semantics of  $(P, A)$ :  $\llbracket (P, A) \rrbracket = \{D \subseteq \mathcal{U} \mid P \subseteq D \cap \mathcal{S} \wedge A \subseteq \mathcal{S} \setminus D\}$ . For instance the present-absent sets  $(\{e\}, \{f\})$  abstract over all databases which contain  $e$  while not containing  $f$ . These present-absent sets can be ordered in a lattice, where smaller sets abstract over more databases, and therefore give more behaviour in the model. This lattice is defined as  $((\mathcal{S} \times \mathcal{S}), \sqsubseteq)$  where  $(P, A) \sqsubseteq (P', A') \iff P' \subseteq P \wedge A' \subseteq A$  and  $(P, A) \sqcup (P', A') = (P \cap P', A \cap A')$ . An example of this lattice with  $\mathcal{S} = \{e\}$  is given in Figure 7.

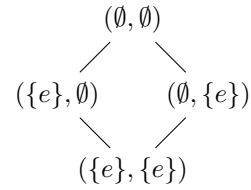


Fig. 7: Lattice example

## A.2 Lattice Transition System

Lattice automata are more easily described in terms of the transition systems they incur.

**Definition 2 (Lattice Transition System).** A lattice transition system (*LaTS*) is a triple  $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$  where  $S$  is a finite set of states,  $\mathcal{L} = (L, \sqsubseteq)$  is a lattice and  $\longrightarrow \subseteq S \times L \times S \times L$  is the transition relation, usually written as  $(s, \ell) \longrightarrow (s', \ell')$  whenever  $(s, \ell, s', \ell') \in \longrightarrow$ , such that for all  $s_1, s_2 \in S$  and  $\ell_1, \ell_2, \ell'_1 \in L$  if  $(s_1, \ell_1) \longrightarrow (s_2, \ell_2)$  and  $\ell_1 \sqsubseteq \ell'_1$  then  $(s_1, \ell'_1) \longrightarrow (s_2, \ell'_2)$  for some  $\ell'_2 \in L$  with  $\ell_2 \sqsubseteq \ell'_2$ .

The behavioural requirement used at the end of the definition is called the *monotonicity property*. Lattice transition systems are a subset of well-structured transition systems. Configurations of an LaTS are pairs of the form  $(s, \ell)$  where  $s \in S$  and  $\ell \in L$  and  $\longrightarrow^*$  denotes the reflexive and transitive closure of  $\longrightarrow$ .

**Definition 3 (Path).** A finite path in an LaTS  $\mathcal{T}$  is a finite sequence  $\sigma = (s_0, \ell_0)(s_1, \ell_1) \cdots (s_n, \ell_n)$  such that  $(s_i, \ell_i) \longrightarrow (s_{i+1}, \ell_{i+1})$  for all  $i$ ,  $0 \leq i \leq n-1$ .

In addition to the standard notion of path we also define *abstracted paths*.

**Definition 4 (Abstracted Path).** An abstracted finite path in an LaTS  $\mathcal{T}$  is a finite sequence  $\hat{\sigma} = (s_0, \ell_0)(s_1, \ell_1) \cdots (s_n, \ell_n)$  such that  $\exists \ell'_{i+1} \in L : (s_i, \ell_i) \longrightarrow (s_{i+1}, \ell'_{i+1})$  and  $\ell'_{i+1} \sqsubseteq \ell_{i+1}$  for all  $i$ ,  $0 \leq i \leq n-1$ .

In the section to follow, we find an efficient way to answer the following question (state-reachability problem): given an initial configuration  $(s_0, \ell_0)$  and a target state  $s_g$ , is there some lattice element  $\ell$  such that  $(s_0, \ell_0) \longrightarrow^* (s_g, \ell)$ ?

## B General Model Checking Algorithm

In Algorithm 1 we present the pseudocode for a general model checking reachability algorithm. The algorithm explores the graph using *waiting* and *passed* sets and depending on the chosen update function performs a different level of abstraction.

It is easy to realize that Algorithm 1 with the simple update implements a search through the whole state space of the lattice transition system. Depending on the way the *waiting* list is organized, it can implement e.g. depth-first search, breath-first search, random search and others. The correctness of this search algorithm is well known.

We now define an alternative update function presented in Algorithms 4 that provides an overapproximation.

### B.1 Join Strategies and CEGAR

The approximative nature of model checking with join update, as described in the previous section, may result in *inconclusive* verification results such that a verification result in the abstract model may not be necessarily realizable in the

---

**Algorithm 1: Reach**( $\mathcal{T}, (s_0, \ell_0), s_g$ )
 

---

**Input:** LaTS  $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$ , initial configuration  $(s_0, \ell_0)$ , a goal state  $s_g \in S$ 
**Output:** “ $s_g$  is reachable” or “ $s_g$  is not reachable”

```

1: waiting :=  $\{(s_0, \ell_0)\}$ 
2: passed :=  $\emptyset$ 
3: while waiting  $\neq \emptyset$  do
4:   Select and remove  $(s, \ell)$  from waiting
5:   passed := passed  $\cup \{(s, \ell)\}$ 
6:   for all  $(s', \ell')$ , where  $(s, \ell) \longrightarrow (s', \ell')$  do
7:     if  $s' = s_g$  then
8:       return “ $s_g$  is reachable”
9:     end if
10:  waiting := Update(passed, waiting,  $(s', \ell')$ )
11: end for
12: end while
13: return “ $s_g$  is not reachable”
    
```

---



---

**Algorithm 2: Update**(*passed*, *waiting*,  $(s, \ell)$ ) \*\*\* Simple Update \*\*\*


---

**Input:** Sets of states *passed* and *waiting* and a configuration  $(s, \ell)$ 
**Output:** Updated set *waiting*

```

1: if  $\exists (s, \ell) \in \textit{waiting} \cup \textit{passed}$  then
2:   return waiting
3: else
4:   return waiting  $\cup \{(s, \ell)\}$ 
5: end if
    
```

---

underlying concrete system. In this case it may be possible to use the lattice structure to derive a more precise approximation that avoids the inconclusive verification result previously encountered.

In this section we describe such an approach to abstraction refinement, inspired by the CEGAR (counter example guided abstraction refinement) principle [2, 3]. The CEGAR approach depends on a few application specific heuristics: a method for determining the feasibility of a path and a method for refining an approximation given an infeasible path. These are formalised in the following.

**Definition 5 (Path feasibility function).** *A path feasibility function determines, in a domain-specific manner, whether an abstracted path is feasible in an LaTS  $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$ :*

$$\textit{pathfeasible} : (S \times L)^* \rightarrow \{\textit{True}, \textit{False}\}$$

The path feasibility function usually corresponds to finding concrete lattice elements for each step in the path, i.e., a concrete path.

Some way of recording the abstractions used in the current state space exploration is needed. At its most abstract this can be viewed as an oracle, answering queries as to whether two lattice elements are allowed to be joined in a given state. We define a joining strategy to capture this notion.

---

**Algorithm 3: Update**( $passed, waiting, (s, \ell)$ )      \*\*\* Cover Update \*\*\*
 

---

**Input:** Sets of states  $passed$  and  $waiting$  and a configuration  $(s, \ell)$   
**Output:** Updated set  $waiting$

- 1: **if**  $\exists(s, \ell') \in waiting \cup passed : \ell \sqsubseteq \ell'$  **then**
- 2:     **return**  $waiting$
- 3: **else**
- 4:     **return**  $waiting \setminus \{(s, \ell') \mid \ell' \sqsubseteq \ell\} \cup \{(s, \ell)\}$
- 5: **end if**

---



---

**Algorithm 4: Update**( $passed, waiting, (s, \ell)$ )      \*\*\* Join Update \*\*\*
 

---

**Input:** Sets of states  $passed$  and  $waiting$  and a configuration  $(s, \ell)$   
**Output:** Updated set  $waiting$

- 1: **if**  $\exists(s, \ell') \in waiting \cup passed : \ell \sqsubseteq \ell'$  **then**
- 2:     **return**  $waiting$
- 3: **else if**  $\exists(s, \ell') \in waiting \cup passed$  **then**
- 4:     **return**  $waiting \setminus \{(s, \ell')\} \cup \{(s, \ell' \sqcup \ell)\}$
- 5: **else**
- 6:     **return**  $waiting \cup \{(s, \ell)\}$
- 7: **end if**

---

**Definition 6 (Joining Strategy).** A joining strategy is a function able to answer questions of the form:

$$strategy_{joining} : S \times L \times L \rightarrow \{True, False\}$$

given an LaTS  $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$ .

The joining strategy can answer that at one state all lattice elements are to be joined, or no elements are to be joined, but it can also answer very selectively which lattice elements to join. In this way the strategy can exploit additional knowledge about the domain: e.g. for integer values, in some parts of the state space the exact value of a variable is needed, in other parts only the parity is relevant, and in yet other parts the signed-ness is important.

The joining strategy need not be perfect. It might give an approximation that leads to an abstracted path to a goal state, which is then deemed infeasible by the path feasibility function. In this situation the strategy is allowed to reconsider some of its answers, at the cost of recomputing the parts of the state space that depended on those answers: a join at some state is allowed to be split. Which state to split at is given by a state split heuristic.

**Definition 7 (State split heuristic).** A state split heuristic determines, in a domain-specific manner, which state to split at, given an infeasible abstracted path in an LaTS  $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$ :

$$h_{splitstate} : (S \times L)^* \rightarrow S$$

**Algorithm 5: CEGAR**( $\mathcal{T}, (s_0, \ell_0), s_g$ )

---

**Input:** LaTS  $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$  (with path feasibility function *pathfeasible*, state split heuristic  $h_{splitstate}$ , and joining strategy *strategyjoining*), initial configuration  $(s_0, \ell_0)$ , a goal state  $s_g \in S$

**Output:** “ $s_g$  is reachable” or “ $s_g$  is not reachable”

- 1:  $waiting := \{(s_0, \ell_0)\}$
- 2:  $passed := \emptyset$
- 3:  $pred := \emptyset$ ; predecessor edges
- 4: **while**  $waiting \neq \emptyset$  **do**
- 5:   Select and remove  $(s, \ell)$  from  $waiting$
- 6:    $passed := passed \cup \{(s, \ell)\}$
- 7:   **for all**  $(s', \ell')$ , where  $(s, \ell) \longrightarrow (s', \ell')$  **do**
- 8:      $pred := pred \cup \{(s', \ell') \rightarrow (s, \ell)\}$ ; record predecessor
- 9:     **if**  $s' = s_g$  **then**
- 10:        $\hat{\sigma} := (s_0, \ell_0) \dots (s_g, \ell')$ ; some abstracted path from  $s_g$  to  $s_0$  in the reverse configuration graph given by vertices  $passed \cup \{(s_g, \ell')\}$  and edges  $pred$
- 11:       **if** *pathfeasible*( $\hat{\sigma}$ ) **then**
- 12:         **return** “ $s_g$  is reachable”
- 13:       **else**
- 14:         ; path was not feasible, due to abstraction
- 15:          $s_{split} := h_{splitstate}(\hat{\sigma})$
- 16:         ; redo exploration from  $s_{split}$
- 17:          $redo := \{(t, \ell') \mid (s_{split}, -) \rightarrow (t, \ell') \in pred\}$
- 18:          $passed := passed \setminus \{(t, -) \mid t \text{ descendant of } s_{split}\}$
- 19:          $waiting := waiting \setminus \{(t, -) \mid t \text{ descendant of } s_{split}\} \cup redo$
- 20:          $pred := pred \setminus \{(-, -) \rightarrow (t, -) \mid t \text{ descendant of } s_{split}\}$
- 21:         **end if**
- 22:       **else**
- 23:         ; add  $(s', \ell')$  to waiting, possibly abstracting by joining
- 24:          $joinelements := \{\ell'' \mid (s', \ell'') \in passed \cup waiting \wedge strategy_{joining}(s', \ell', \ell'') = True\}$
- 25:          $\ell_{joined} := \ell' \sqcup (\bigsqcup joinelements)$
- 26:          $pred := pred \cup \{(s', \ell_{joined}) \rightarrow (t, \ell''') \mid \ell'' \in joinelements \text{ s.t. } (s', \ell'') \rightarrow (t, \ell''') \in pred\}$
- 27:          $passed := passed \setminus \{(s', \ell'') \mid \ell'' \in joinelements\}$
- 28:          $waiting := waiting \cup \{(s', \ell_{joined})\} \setminus \{(s', \ell'') \mid \ell'' \in joinelements\}$
- 29:       **end if**
- 30:   **end for**
- 31: **end while**
- 32: **return** “ $s_g$  is not reachable”

---

We can now present the complete algorithm for CEGAR exploration of a LaTS in Algorithm 5. The explorations will start from an abstraction level close to that of the join update (depending on the joining strategy) and become less abstract until it at some point becomes the same as the cover update, unless a conclusive answer is found before that.