opaal: A Lattice Model Checker

Andreas Engelbredt 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
{andrease,rrh,kyrke,kgl,mchro,petur,srba}@cs.aau.dk

Abstract. We present a new open source model checker, opaal, 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 opaal 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 opaal, and demonstrate how opaal 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 a given verification purpose are "abstracted away".

The opaal model checker described in this paper allows for such abstractions to be integrated in the model through user-defined lattices. Models are formalised by *lattice automata*: 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*. The addition of explicit lattices makes it possible to apply some of the advanced concepts and expressive power of abstract interpretation directly in the models.

Lattice automata, as implemented in opaal, 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 have a *join operator* that joins two lattice elements by computing their least upper bound, thereby potentially overapproximating the behaviour, with the gain of a reduced state space. Model checking the overapproximated model can however

M. Bobaru et al. (Eds.): NFM 2011, LNCS 6617, pp. 487–493, 2011.

be inconclusive. We introduce the notion of a *joining strategy* affording the user more control over the overapproximation, by specifying which lattice elements are joinable. This allows for a form of user-directed CEGAR (Counter-Example Guided Abstraction Refinement) [2,3]. The CEGAR approach can easily be automated by the user, by exploiting application-specific knowledge to derive more fine-grained joining strategies given a spurious error trace. Thus providing, for some systems and properties, efficient model checking and conclusive answers at the same time.

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

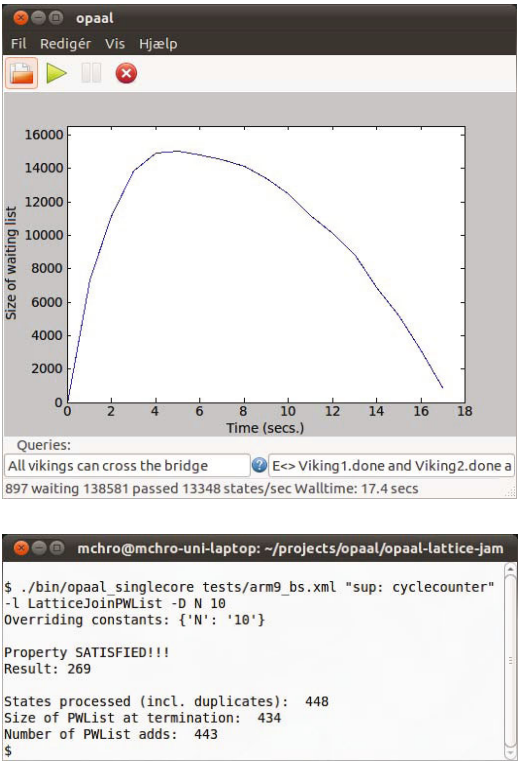


Fig. 1. `opaal` GUI and CLI

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 new lattices by implementing simple Python class interfaces. The new classes can then be used directly in the model (including all user-defined methods). Joining strategies are defined as Python functions.

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".

The "Successor Generator" exposes an interface that the "Reachability Checker" can use to perform the actual verification. During this process a

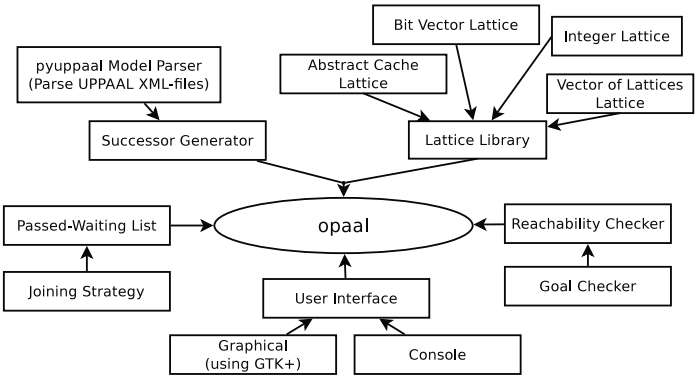


Fig. 2. Overview of `opaal`'s architecture

"Passed-Waiting List" is used to save explored and to-be explored states; it employs a user-provided "Joining Strategy" on the lattice elements of states, before they are added to the list.

2 Examples

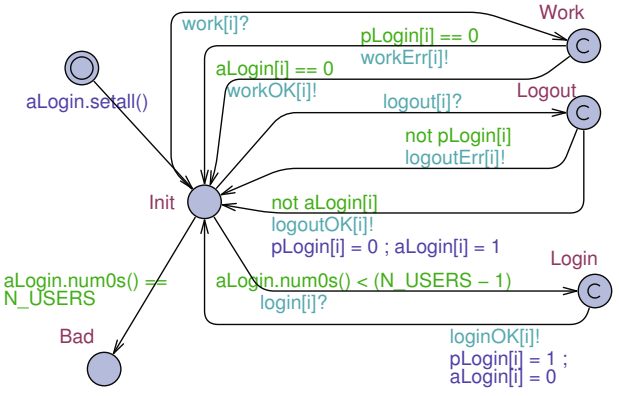
In this section we present a few examples to demonstrate the wide applicability of `opaal`. The tool currently has a number of readily available lattices that are used to abstract the real data in our examples.

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.




```
1 const int N_USERS = 17;
2 ...
3 extern IntersBitVector;
4 IntersBitVector pLogin[N_USERS];
5 IntersBitVector aLogin[N_USERS];
```

```
1 class IntersBitVector:
2     def join(self, other):
3         ...
4
5     def __le__(self, other):
6         ...
```

Fig. 3. (a) Database model (b) Lattice variables (c) 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.

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. 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 it sent, it is added to the set *pool* and it remains

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"

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

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 `opaal`. Here we have N agents with their unique identifications $0, 1, \ldots, N-1$ 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 model checker, `opaal`, for lattice automata and provided a number of applications. The expressiveness of the formalism, derived from well-structured transition systems, promises 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), 63–92 (2001)
2. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002, pp. 58–70. ACM, New York (2002)
3. Ball, T., Rajamani, S.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (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 (2010)
6. Garcia-Molina, H.: Elections in a distributed computing system. IEEE Trans. Comput. 31(1), 48–59 (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., Stenstrm, P.: The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. Trans. on Embedded Comp. Sys. 7(3), 1–53 (2008)

8. Alt, M., Ferdinand, C., Martin, F., Wilhelm, R.: Cache Behavior Prediction by Abstract Interpretation. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 52–66. Springer, Heidelberg (1996)
9. Henzinger, T., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and Computation 111(2), 193–244 (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. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)