Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems

Tool Paper

Sylvain Conchon[1], Amit Goel[2], Sava Krstić[2], Alain Mebsout[1], and Fatiha Zaïdi[1]

[1] LRI, Université Paris Sud, CNRS, Orsay F-91405
[2] Strategic CAD Labs, Intel Corporation

Abstract. Cubicle is a new model checker for verifying safety properties of parameterized systems. It implements a parallel symbolic backward reachability procedure using Satisfiabilty Modulo Theories. Experiments done on classic and challenging mutual exclusion algorithms and cache coherence protocols show that Cubicle is effective and competitive with state-of-the-art model checkers.

1 Tool Overview

Cubicle is used to verify safety properties of *array-based systems*. This is a syntactically restricted class of parametrized transition systems with states represented as arrays indexed by an arbitrary number of processes [10]. Cache coherence protocols and mutual exclusion algorithms are typical examples of such systems. Cubicle model-checks by a symbolic backward reachability analysis on infinite sets of states represented by specific simple formulas, called *cubes*.

Cubicle is an open source software based on theoretical work in [1] and [11]. It is inspired by and closely related to the model checker MCMT [12], from which, in addition to revealing the implementation details, it mainly differs in a more friendly input language and concurrent architecture.

Cubicle is written in OCaml. Its SMT solver is a tightly integrated, lightweight and enhanced version of Alt-Ergo [7]; and its parallel implementation relies on the Functory library [9]. Cubicle is available at `http://cubicle.lri.fr`.

2 System Description Language

Cubicle's input language is a typed version of Murφ [8] similar to the one of UCLID [6], rudimentary at the moment, but more user-friendly than MCMT and sufficiently expressive for typical parameterized systems.

A system is described in Cubicle by: (1) a set of type, variable, and array declarations; (2) a formula for the initial states; and (3) a set of transitions. It is parametrized by a set of *process identifiers*, denoted by the built-in type `proc`. Standard types `int`, `real`, and `bool` are also built in. Additionally, the user

P. Madhusudan and S.A. Seshia (Eds.): CAV 2012, LNCS 7358, pp. 718–724, 2012.

can specify abstract types and enumerations with simple declarations like "`type data`" and "`type msg = Empty | Req | Ack`". We show the language on the following `Mutex` example.

```
var Turn : proc
array Want[proc] : bool
array Crit[proc] : bool

init (z) {
  Want[z] = False && Crit[z] = False
}

unsafe (x y) {
  Crit[x] = True && Crit[y] = True
}

transition req (i)
requires { Want[i] = False }
{ Want[j] := case
           | i = j : True
           | _ : Want[j]  }
```

```
transition enter (i)
requires { Want[i] = True
        && Crit[i] = False
        && Turn = i }
{ Crit[j] := case
           | i = j : True
           | _ : Crit[j]  }

transition exit (i)
requires { Crit[i] = True }
{ Turn := . ;
  Crit[j] := case
           | i = j : False
           | _ : Crit[j]  ;
  Want[j] := case
           | i = j : False
           | _ : Want[j]  }
```

The system's state is defined by a set of global variables and `proc`-indexed arrays. The initial states are defined by a universal conjunction of literals characterizing the values for some variables and array entries. A state of our example system `Mutex` consists of a process identifier `Turn` and two boolean arrays `Want` and `Crit`; a state is initial iff both arrays are constantly `false`.

Transitions are given in the usual guard/action form and may be parameterized by (one or more) process identifiers. They define the system's execution: an infinite loop that at each iteration: (1) non-deterministically chooses a transition instance whose guard is true in the current state; and (2) updates state variables according to the action of the fired transition instance. Guards must be of the form $F \wedge \forall \bar{x}.(\Delta \Rightarrow F')$, where F, F' are conjunctions of literals (equations, disequations or inequations), and Δ says that every $\bar{x}$-variable is distinct from every parameter of the transition. Assignments can be non-deterministic, as in "`Turn := .`" in transition `exit` in `Mutex`. Array updates are coded by a case construct where each condition is a conjunction of literals, and the default case.

The safety property to be verified is expressed in its negated form as a formula that represents unsafe states. Each unsafe formula must be a *cube*, i.e., have the form $\exists \bar{x}.(\Delta \wedge F)$, where Δ is the conjunction of all disequations between the variables in $\bar{x}$, and F is a conjunction of literals. In the code, we leave the Δ part implicit. Thus in `Mutex`, the unsafe states are those in which `Crit[x]` and `Crit[y]` are true for two distinct process identifiers `x,y`.

3 Implementation Details and Optimizations

For a state formula Φ and a transition instance t, let $pre_t(\Phi)$ be the formula describing the set of states from which a Φ-state can be reached in one t-step.

Let also $pre(\Phi)$ be the union of $pre_t(\Phi)$ for all possible t. In its simplest form, the backward reachability algorithm constructs a sequence $\Phi_0, \Phi_1, \ldots$ such that Φ_0 is the system's unsafe condition and $\Phi_{i+1} = \Phi_i \vee pre(\Phi_i)$. The algorithm terminates with the first Φ_n that fails the *safety check* (consistency with the initial condition), or passes the *fixpoint check* $\Phi_n \vdash \Phi_{n-1}$.

In array-based systems, $pre_t(\phi)$ can be representend as a union (disjunction) of cubes, for every cube ϕ and every t. Thus, the Φ_i above are unions of cubes too, and the algorithm above can be modified to work only with cubes, as follows. Maintain a set V and a priority queue Q of *visited* and *unvisited cubes* respectively. Initially, let V be empty and let Q contain the system's unsafe condition. Then, at each iteration, take the highest-priority cube ϕ from Q and do the safety check for it, same as the above. If it fails, terminate with "system unsafe". If the safety check passes, proceed to the *subsumption check* $\phi \vdash \bigvee_{\psi \in V} \psi$. If this fails, then add ϕ to V, compute all cubes in $pre_t(\phi)$ (for every t), add them to Q, and move on to the next iteration. If the subsumption check succeeds, then drop ϕ from consideration and move on. The algorithm terminates when a safety check fails or Q becomes empty. When an unsafe cube is found, Cubicle actually produces a counterexample trace.

Safety checks, being ground satisfiablity queries, are easy for SMT solvers. The challenge is in subsumption checks $\phi \vdash \bigvee_{\psi \in V} \psi$ because of their size and the "existential implies existential" logical form. Assuming $\phi \triangleq \exists \bar{x}.F$ and $\psi \triangleq \exists \bar{y}.G_\psi$ ($\psi \in V$), the subsumption check translates into the validity check for the ground formula $H \triangleq (F \Rightarrow \bigvee_{\psi \in V} \bigvee_{\sigma \in \Sigma} (G_\psi)\sigma)$, where Σ is the set of all substitutions from $\bar{y}$ to $\bar{x}$. Now, viewing any cube $G_\psi\sigma$ as a set of literals, one can make two useful comparisons with F: (1) if $G_\psi\sigma$ is a subset of F, then H is valid; (2) if $G_\psi\sigma$ contains a literal that directly contradicts a literal of F, then $G_\psi\sigma$ is redundant in H (can be removed without logically changing H). Cubicle aggressively attempts to prove H by building and verifying it incrementally, adding one disjunct to its consequent at a time. Essentially, it examines all pairs (ψ, σ) one-by-one, stopping the process when the current overapproximation of H becomes known to be valid. For each pair (ψ, σ), the cube $G_\psi\sigma$ is first checked for redundancy; if redundant, it is ignored and a new pair (ψ, σ) is processed. If not redundant, the cube is subject to the subset check for $F \vdash G_\psi\sigma$. If this check succeeds, H is claimed valid; otherwise $G_\psi\sigma$ gets added to H (as a disjunct of its consequent) and the SMT solver checks if the newly obtained (weakened) H becomes valid.

Cubicle's integration with the SMT solver at the API level is crucial for efficient treatment of the subsumption check. For any such check, a single context for the SMT solver is used; it just gets incremented and repeatedly verified. To support the efficient (symmetry-reduced) and exhaustive application of the inexpensive redundancy and subset checks, cubes are maintained in normal form where variables are renamed and implied literals removed at construction time.

The strategy for exploring the cube space is also essential. It pays to visit as few cubes as possible, which suggess giving priority to more "generic" cubes (those that represent larger sets of states). Thus, neither breadth-first nor depth-first search are good in their pure form. By default, Cubicle uses BFS (changeable

with the `-search` option to DFS or some variants) combined with a heuristically delayed treatment of some cubes. Currently, a cube is delayed if it introduces new process variables or does not contribute new information on arrays. Finally, Cubicle can remove cubes from $\mathcal{V}$ when they become subsumed by a new cube.

Following MCMT, Cubicle supports user-supplied invariants and invariant synthesis, both of which can significantly reduce the search. Subsets of visited nodes that only contain predicates over a unique process variable are used as candidate invariants. Each of them is verified by starting a new resource limited backward reachability analysis. Cubicle can also discover "subtyping invariants" (saying that a variable can take only a selected subset of values) by a static analysis and these invariants can be natively exploited by the SMT solver which supports definitions of subtypes for enumerated data-types.

4 Multi-core Architecture

A natural way to scale up model checkers is to parallelize their CPU intensive tasks to take advantage of the widespread availability of multi-core machines or clusters [13,4,14]. In our framework, this is achieved by parallelizing the backward reachability loop and the generation of invariants. As mentioned above, since invariant synthesis is done independently from the main loop, it is straightforward to do it in parallel. However, concerning the loop itself, a naive parallel implementation would lose the precise guidance of the exploration[1], and more importantly, could break the correctness of the tool because of an unsafe use of some optimizations described in the previous section.

In our setting, the most resource consuming tasks are fixpoints checks which can be hard problems even for efficient SMT solvers. To gain efficiency, we implemented a concurrent version of BFS based on the observation that all such computations arising at the same level of the search tree can be executed in parallel. Our implementation is based on a centralized master/workers architecture. The master assigns fixpoints to workers and a synchronization barrier is placed at each level of the tree to retain a BFS order. The master asynchronously computes the preimages of nodes that are not verified as fixpoints by the workers. In the meanwhile, the master can also assign invariant generation tasks that will be processed by available workers. Finally, to safely delete nodes from $\mathcal{V}$, the master must discard the results about nodes that have been deleted while they were being checked by a worker.

Cubicle provides a concurrent breadth-first exploration of the search space using n parallel processes on a multi-core machine with the `-j` n option. The implementation is based on Functory [9], an OCaml library with a rich functional interface which facilitates the execution of parallel algorithms. Functory supports multi-core architectures and distributed networks; it has also a robust fault-tolerance mechanism. Concerning a distributed implementation, one of the main issues is to limit the size of data involved in transactions between the master and

[1] Our experiments showed that a non-deterministic parallel exploration can be worse than a guided sequential search.

the workers. For instance, the size of $\mathcal{V}$ can quickly become a bottleneck in an architecture based on message passing communications. As future work, we plan to develop a distributed implementation that will only need to send updates of data-structures.

5 Experimental Results and Future Works

We have evaluated Cubicle on some classic and challenging mutual exclusion algorithms and cache coherence protocols. In the table bellow, we compare Cubicle's performances with state-of-the-art model checkers for parameterized systems. All benchmarks have been executed on a 64 bits machine with a quad-core Intel® Xeon® processor @ 3.2 GHz and 24 GB of memory. For each tool, we report the results obtained with the best settings we found. Note that the parallel version of Cubicle was run on 4 cores and that we only give its results for significantly time consuming problems. We denote by X benchmarks that we were unable to translate due to syntactic restrictions.

	Cubicle		MCMT [12]	Undip [3]	PFS [2]
	seq	4 cores			
bakery	0.01s	-	0.01s	0.04s	0.01s
Dijkstra	0.24s	-	0.99s	**0.04s**	0.26s
Distributed_Lamport	**2.3s**	-	12.7s	unsafe	X
Java_Mlock	0.04s	-	0.06s	0.25s	0.02s
Ricart_Agrawala	**1.8s**	-	1m12s	4.3s	X
Szymanski_at	**0.12s**	-	0.71s	13.5s	timeout
Berkeley	0.01s	-	0.01s	0.01s	0.01s
flash_aggregated [15]	0.01s	-	0.02s	0.01s	X
German_Baukus	**25.0s**	17.1s	3h39m	9m43s	X
German_pfs	**6m23s**	3m8s	11m31s	timeout	47m22s
German_undip	**0.17s**	-	0.57s	1m32	X
Illinois	0.02s	-	0.04s	0.06s	0.06s
Moesi	0.01s	-	0.01s	0.01s	0.01s

Our experiments are very promising. They show first that the sequential version of Cubicle is competitive. The parallel version on 4 cores achieves speedups of 1.8 approximately, which is a good result considering the fact that cores cannot be fully exploited because of the synchronization required to perform a pertinent search. In practice, we found that the best setting for Cubicle is to use all the optimizations described in Section 3 (except for invariant synthesis which can be time consuming). In the table bellow, we show the respective effect of these optimizations on the version of the German protocol from [5] (German_baukus). In particular, it is worth noting that the subtyping analysis increases performances by an order of magnitude on this example.

Optimizations			**Real Time** (# nodes)	
delete nodes	subtyping	invariant generation	sequential	4 cores
No	No	No	50m8s (22580)	27m13s (20710)
Yes	No	No	35m16s (20405)	19m39s (19685)
Yes	No	Yes	20m45s (15089)	13m55s (14527)
Yes	Yes	No	**25.0s** (3322)	**17.1s** (3188)

As future work, we would like to harness the full power of the SMT solver by sharing its data structures and even more tightly integrating its features in the model checker. In particular, this would be very useful to discover symmetries and to simplify nodes by finding semantic redundancies modulo theories. We are also interested in exploiting the unsat cores returned by the solver to improve our node deletion mechanism.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321 (1996)
2. Abdulla, P.A., Delzanno, G., Ben Henda, N., Rezine, A.: Regular Model Checking Without Transducers (On Efficient Verification of Parameterized Systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
3. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized Verification of Infinite-State Processes with Global Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
4. Barnat, J., Brim, L., Češka, M., Ročkai, P.: DiVinE: Parallel Distributed Model Checker (Tool paper). In: HiBi/PDMC, pp. 4–7 (2010)
5. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 317–330. Springer, Heidelberg (2002)
6. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 78–92. Springer, Heidelberg (2002)
7. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC(X): Semantic combination of congruence closure with solvable theories. ENTCS 198(2), 51–69 (2008)
8. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: ICCD, pp. 522–525 (1992)
9. Filliâtre, J.-C., Kalyanasundaram, K.: Functory: A distributed computing library for Objective Caml. In: TFP, pp. 65–81 (2011)
10. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. In: IJCAR, pp. 67–82 (2008)
11. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. LMCS 6(4) (2010)
12. Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: IJCAR, pp. 22–29 (2010)

13. Grumberg, O., Heyman, T., Ifergan, N., Schuster, A.: Achieving Speedups in Distributed Symbolic Reachability Analysis Through Asynchronous Computation. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 129–145. Springer, Heidelberg (2005)
14. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in Eddy. STTT 11(1), 13–25 (2009)
15. Park, S., Dill, D.L.: Protocol Verification by Aggregation of Distributed Transactions. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 300–310. Springer, Heidelberg (1996)