MERIT: An Interpolating Model-Checker

Nicolas Caniart

LaBRI, Université de Bordeaux - CNRS UMR 5800
caniart@labri.fr

Abstract. We present the tool MERIT, a CEGAR model-checker for safety properties of counter-systems, which sits in the Lazy Abstraction with Interpolants (LAWI) framework. LAWI is parametric with respect to the interpolation technique and so is MERIT. Thanks to its open architecture, MERIT makes it possible to experiment new refinement techniques without having to re-implement the generic, technical part of the framework. In this paper, we first recall the basics of the LAWI algorithm. We then explain two heuristics in order to help termination of the CEGAR loop: the first one presents different approaches to symbolically compute interpolants. The second one focuses on how to improve the unwinding strategy. We finally report our experimental results, obtained using those heuristics, on a large amount of classical models.

1 Motivations

Lazy Abstraction with interpolants (LAWI). [8] is a generic technique to verify the safety of a system. It builds a tree by unwinding the control structure of the system. Each edge of this tree represents a transition between two control points, and each node is labeled by an over-approximation of the actual reachable configuration at that node.

LAWI follows the CEGAR [3] paradigm. It loops over three steps: *explore*, *check*, and *refine*. The *explore* step expands the reachability tree by unwinding the control structure. At first, one starts from a very coarse abstraction of the system, ignoring the transition effect, just checking the reachability of control locations marked as bad. For that reason, reaching a bad location does not necessarily mean that the system is unsafe. The *check* step looks at a branch leading to a bad location, and tries to prove that it is spurious, that is, not feasible in the actual system. If it fails, then the system is unsafe and an error trace is reported. If it succeeds, then the unwinding must be refined to eliminate this spurious path. The *refine* step consists in labeling each node on the branch by an *interpolant* that over-approximates more closely the actual configurations. This explains the term *lazy* [5]: the refinement occurs only on a branch, not on the whole tree.

Since the unwinding is in general infinite, the algorithm might not terminate. To help termination, LAWI uses a *covering* relation between nodes. Under some conditions one can guarantee that any configuration reachable from a node s is also reachable from a node t in the tree. Node t then covers node s, which prevents from having to explore the subtree rooted at s, thus limiting the tree growth. LAWI terminates when all leaves in the unwinding tree are covered. However, since nodes are relabeled during the refine step, a covered node may be uncovered, so that termination can still not be guaranteed.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 162–166, 2010.

Interpolation with Symbolic Computation. Let us explain more precisely how to refine a spurious path $s_0 \xrightarrow{\tau_1} s_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} s_n$, where each s_i is a node of the unwinding and each τ_i a transition of the system. An interpolant for this path [8] is given by sets $I_0, I_1, \ldots, I_n$ such that I_0 over-approximates the initial set of states, $\tau_k(I_k) \subseteq I_{k+1}$ for any k and $I_n = \emptyset$. Such a path interpolant witnesses that the path is spurious. In general, there are several path interpolants. The choice of the interpolant affects the algorithm behavior (and termination), since the covering relation depends on it.

In [8] theorem proving and Craig interpolation is used to compute interpolants. Our work focuses on model-checking counter-systems [9] with unbounded variables. The transition relations are encoded in the Presburger logic. So far, Craig interpolation algorithms are known only for fragments of this logic [6]. In [4] it is showed how to compute interpolants symbolically. We have chosen to first experiment symbolic computation techniques in our model-checker, using the TaPAS tools [7]: an interface to various automata- or formula-based symbolic set representations.

2 The MERIT Model-Checker

MERIT is a model-checker for counter-systems based on the LAWI framework. We discuss its architecture, and present two improvements to the generic algorithm that we both implemented in MERIT.

Open architecture. An interesting feature of the LAWI algorithm is that it offers a clear split between operations that work only on the control graph of the transition system, and those that compute interpolants. Thanks to this, we were able to use virtually any kind of techniques to compute interpolants: theorem proving, SMT-solvers, or symbolic computation. All operations or queries made on interpolants are implemented behind the interface of a single module, called a *refinery*. Changing the way interpolants are computed is just a matter of changing the refinery. We currently have one fully functional refinery based on the TAPAS framework [7] and we are working on an SMT-solver based refinery.

Optimizing the interpolants. MERIT implements the classical symbolic weakest pre- and strongest post-conditions computations, which provide path interpolants [4]. They are named weakest (resp. strongest) since they are the maximal (resp. minimal) sets of configurations that can appear on a path interpolant. MERIT also implements two original, and in practice more efficient techniques, that we both experimented.

- The first uses post- symbolic computation to find the closest node from the root of the refined branch where the reachability set becomes empty. From that node, it replays the trace backwards, computing the weakest path interpolant. This way, we obtain shorter branches than when using a weak-pre computation, but weaker interpolants on the higher part of the branch than when using a post computation. The idea of the heuristic is to make it easier to cover nodes by producing large interpolants high in the tree.
- The second one is the dual of the first: it starts with a backward symbolic computation, and it then tightens interpolants on the lower part of the branch using strongest interpolants.

Experimental results for the later technique, called *cut-post*, are given in Table 1.

Tuning the unwinding strategy: BFS vs. DFS. Our experiments also show that the strategy used to expand the tree has an impact on the algorithm termination. In [8] it is suggested to use a DFS strategy to expand the tree. Indeed a DFS strategy seems more adequate than a BFS one: suppose that the algorithem has to visit nodes at depth d. With systems having control locations of out-degree k, it is then necessary to compute and store at least k^d nodes. In practice models with few control locations and high out-degree emerge naturally[1].

The problem comes from the fact that a naive implementation of the DFS expansion strategy can behaves like a BFS. Indeed, in [8] a node is expanded by adding all its children at once: e.g. a node t gets 3 children u, v, w. Because those nodes have not been refined, they are labeled by the full variable domain. Thus each of them can cover any node added after itself (provided they correspond to the same control location). The DFS proceeds by expanding u. Suppose now that children of u are all covered by either v or w. The DFS is therefore stopped and we have to explore a new branch (in v subtree). Again, the children of v may become covered by w. We see how a "BFS-like" behavior arises. This phenomenon does occur in practice and a combinatorial blowup indeed impairs the algorithm termination.

To fix this problem, we add only one child, and pursue the DFS with it. When that child is popped from the DFS stack, we add its sibling and repeat the same process. This way we add less nodes labeled by the full variable domain, which prevents from covering uselessly. The termination condition becomes more complicated. We also have to check that we did not forget to fully expand all internal nodes. Nonetheless our experiments show that, using this strategy, our tool can cope with models where the original strategy fails. The impact on the tool performance can be drastic, as showed in Table 1.

3 Experimental Results

MERIT has been tested with a pool of about 50 infinite-state systems, ranging from broadcast protocols to programs with dynamic data-structures. The benchmark suite we use is available on the tool webpage (cf. Availability section, p. 165). The verification was successful in about 80% of the tests and MERIT detected 100% of the unsafe models.

The use of the "append one child at a time" unwinding strategy and the cut-pre or cut-post refinement techniques presented in Sec. 2 allowed MERIT to almost double the number of models it can tackle. Table 1 presents the results obtained (1) with the original algorithm, the weakest pre-conditions refinement (column Original Pre), the one child at a time algorithm with the same refinement technique (column 1 child Pre), as well as the one child algorithm with the cut-post refinement method (column 1 child cut-post). This shows how much we improved from the original algorithm. We also compare our tool to the tools FAST[2] and ASPIC[3] because they make use of acceleration [1] techniques which are particularly efficient on the models we use to test MERIT. However MERIT is more efficient than FAST and ASPIC on many models.

[1] Like for distributed system models, where the global control structure encoded by variables.

[2] Available at `http://www.lsv.ens-cachan.fr/Software/fast/`

[3] Available at `http://laure.gonnord.org/pro/aspic/aspic.html`

Table 1. Benchmark results

				Original Pre			1 child Pre			1 child cut-post			FAST	ASPIC
MODEL	V	T	O	N	R	TIME	N	R	TIME	N	R	TIME	TIME	TIME
ILLINOIS	5	9	S	4152	415	2.12	777	388	1.72	-	-	TOUT	1.75	?
insert	48	51	S	74	11	1.28	70	11	1.35	70	11	1.25	3.97	0.14
MESI	5	4	S	287	57	1.42	107	53	1.05	35	17	1.13	1.71	?
merge	847	1347	S	6661	944	27.77	5413	952	40.34	189	30	3.79	TOUT	2.27
MOESI	6	4	S	27	5	1.23	11	5	1.14	35	17	1.16	1.36	?
train	7	12	S	20878	4302	268.64	205	101	1.51	1531	765	13.55	2.29	?
deleteAll	18	19	U	13	2	1.10	13	2	0.98	13	2	1.18	1.0	0.11

Legend: V = # of variables; T: # of transitions; O: outcome, S means safe, U unsafe, ? tool does not know ; N: # tree nodes, R: # refinements. TOUT means time-out, MOUT memory outage.

4 Conclusion and Development Perspectives

In this paper we presented MERIT, a model-checker tool that use symbolic interpolant computation techniques. It implements the Lazy Abstraction with Interpolants algorithm [8]. The models we experimented are particularly suited for acceleration techniques. However MERIT was able to tackle many models without using acceleration.

Short-term goals: One of our short term goals is to get a fully functionnal SMT-Solver based refinery, to see how such a technique can compete with symbolic ones.

Mid-term goals: We noticed that some refinement techniques are complementary: they succeed on different sets of models and the union of those sets almost covers the whole set of models. We tried hybrid refinement techniques that combine them. This allowed MERIT tackle more models. However the problem of choosing, on the fly, the proper interpolation technique for a branch is still an open problem.

Finally, our experiments showed that some difficult examples would benefit from *acceleration* [2] techniques like the `train` model in Tab. 1. However combining LAWI and acceleration is still an open question. Acceleration is costly and the trade-off between that cost and the benefit for the cover relation has to be investigated.

Availability. MERIT is available under free software license at `http://www.labri.fr/~caniart/merit.html`.

References

[1] Boigelot, B.: On Iterating Linear Transformations Over Recognizable Sets of Integers. Theoretical Computer Science 309(1-3), 413–468 (2003)

[2] Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 428–443. Springer, Heidelberg (2008)

[3] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)

[4] Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with Craig interpolation and symbolic pushdown systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 489–503. Springer, Heidelberg (2006)
[5] Henzinger, T.A., Jhala, R., Majumbar, R., Sutre, G.: Lazy Abstraction. In: Proc. of POPL'02, pp. 58–70 (2002)
[6] Jain, H., Clarke, E.M., Grumberg, O.: Efficient Craig interpolation for linear diophantine (dis)equations and linear modular equations. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 254–267. Springer, Heidelberg (2008)
[7] Leroux, J., Point, G.: Tapas: The Talence Presburger Arithmetic Suite. In: Proc. of TACAS'09. LNCS, vol. 5505, pp. 182–185. Springer, Heidelberg (2009)
[8] McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
[9] Minsky, M.L.: Computation: Finite and Infinite Machines, June 1967. Prentice-Hall, Englewood Cliffs (June 1967)