Merit: an Interpolating Model-Checker

Nicolas Caniart

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.

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 $I_n = \varnothing$. 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 sub-tree). 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 at a time. 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. 4). 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.

4 Conclusion & 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 algo-

[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

				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.

Table 1. Benchmark results

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

[2] N. Caniart, E. Fleury, J. Leroux, and M. Zeitoun. Accelerating interpolation-based model-checking. In *Proc. of TACAS'08*, vol. 4963 of *LNCS*, pages 428–443. 2008.

[3] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV'00*, vol. 1855 of *LNCS*, pages 154–169. 2000.

[4] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *Proc. of TACAS'06*, *LNCS*, pages 489–503. 2006.

[5] T. A. Henzinger, R. Jhala, R. Majumbar, and G. Sutre. Lazy Abstraction. In *Proc. of POPL'02*, pages 58–70. 2002.

[6] H. Jain, E. M. Clarke, and O. Grumberg. Efficient Craig interpolation for linear diophantine (dis)equations and linear modular equations. In *Proc. of CAV'08*, vol. 5123 of *LNCS*, pages 254–267. 2008.

[7] J. Leroux and G. Point. Tapas : The Talence Presburger Arithmetic Suite. In *Proc. of TACAS'09*, vol. 5505 of *LNCS*, pages 182–185. 2009.

[8] K. L. McMillan. Lazy Abstraction with Interpolants. In *Proc. of CAV'06*, vol. 4144 of *LNCS*, pages 123–136. 2006.

[9] M. L. Minsky. *Computation : Finite and Infinite Machines*. Prentice Hall, June 1967.

The presentation of the tool will be conducted around the model we use in the example section below.

A Availability

MERIT is available under free software license and can be downloaded from http://www.labri.fr/~caniart/merit.html. The software is built on top of many libraries. For this reason we also provide pre-built binaries for the Linux platform.

Benchmarks The results for the complete benchmarks suite we use is provided at http://www.labri.fr/~caniart/benchmark-cav10.html.

B Merit input

Merit input format is the same as the one of the tool Fast[4]. Briefly, this format consists of two parts: a description of the model through the declaration of a set of variables, control locations and transitions; a strategy in which a little script language allows to define sets of states (regions), test for set intersection, and parametrize Fast behaviour. It will ignore anything else. Fast syntax only allows for linear constraints in guards, actions (updates) and region definitions. The sample below sums up the fragment of the FAST format syntax that MERIT understands, as well as the formal grammar for the linear constraints that can be used in FAST guards, updates and regions definitions.

```
1  model ModelName {
2    var    ID0, ID1, ID2, ... ;
3    states LOC0, LOC1, ... ;
4
5    transition t0 := {
6      from   := <state> ;
7      to     := <state> ;
8      guard  := <expr> ;
9      action := <updatelist> ;
10   } ;
11
12   ...
13 }
14
15 strategy StrategyName {
16   Region ID := { <region> } ;
17 }
```

$$
\begin{aligned}
region &:= region \parallel expr \mid region \,\&\&\, expr \mid \\
&\quad region \parallel statepred \mid \\
&\quad region \,\&\&\, statepred \mid \\
&\quad statepred \mid expr \\
expr &:= pred \mid expr \,\&\&\, pred \mid expr \parallel pred \mid \\
&\quad (\ expr\ ) \\
statepred &:= \text{state} = \text{LOC} \\
pred &:= lsum \ \# \ lsum \\
updatelist &:= update \mid updatelist\ ,\ update \\
update &:= \text{VAR}' \ = \ lsum \\
lsum &:= term \mid lsum \ + \ term \mid lsum \ - \ term \\
term &:= \text{VAR} \mid \text{NUMBER} \mid \text{NUMBER} \ \times \ \text{VAR}
\end{aligned}
$$

Names between < > in the input sample refer to the corresponding rule in the grammar. In the grammar, $\# \in \{\leq, <, =, >, \geq\}$, VAR refers to a variable declared in the `var` statement (l. 2) and LOC to a location name declared in the `state` statement (l. 3). The complete reference of the Fast input format syntax is provided in the Fast User Manual[2]. Figure 1 provides the complete description of a model in FAST syntax.

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

Input processing The input is parsed by MERIT to build the transition system augmented with one initial and one error location (resp. denoted '.' and '!'). Those locations are connected to the transition system according the INIT and ERROR regions. MERIT looks for those two regions in the strategy; they respectively define the sets of initial and bad states. The INIT set is used as a guard on the edge that leaves the '.' location and connects it to the original transition system. This transition is denoted INIT?. Similary the ERROR region is used on the ERROR? transition that connects the original transition system to the '!' location[5].

C Interesting points on a running example

```
model MESI {
  // m: modified, e: exclusive, s: shared, i: invalid.
  var    m, e, s, i ;
  states s ;

  transition ReadInv := {
    from   := s ;
    to     := s ;
    guard  := i >= 1 ;
    action := m'= 0, e'= 0,
              s'= m+e+s+1 ;
              i'= 0 ;
  } ;

  transition Share := {
    from   := s ;
    to     := s ;
    guard  := e >= 1 ;
    action := m' = m + 1,
              e' = e - 1 ;
  } ;

  transition WriteShared := {
    from   := s ;
    to     := s ;
    guard  := s >= 1 ;
    action := m' = 0, e' = 1,
              s' = 1,
              i' = m+e+s+i-1 ;
  } ;

  transition Load := {
    from   := s ;
    to     := s ;
    guard  := i >= 1 ;
    action := m' = 0, e' = 1,
              s' = 0,
              i' = m+e+s+i-1 ;
  } ;
}

strategy MESIStrategy {
  Region INIT  := { m=0 && e=0 && s=0 && i>0 } ;
  Region ERROR := { m>=2 || ( m>=1 && s>=1 ) } ;
}
```

Fig. 1. Model of the MESI Cache Coherence Protocol in Fast language

We now present, on a simple model, how the tuning in the exploration and refinement processes improves the original algorithm. We will look at a model of the MESI cache coherence protocol. Its description in FAST language is given on fig. 1. The corresponding transition system, augmented with the '.' and '!' locations, is depicted on fig. 2(b).

First, to give an intuition of how LAWI works, we will go through a few iterations of the three algorithm steps: explore, check, and refine. Then we explain in more details the issue in the original unwinding strategy we talked about in sec. 2. Finally we will discuss the influence of the interpolant computation techniques.

[5] More than one transition may actually leave locations '.' and '!'. So MERIT appends an index to those transitions names. This will not be the case in the forthcoming example, and we ignore those indices here.

C.1 How LAWI works

LAWI first step, *explore* is responsible for unwinding the transition system. Explore proceeds as follows: it picks a node in the unwinding tree. It first tries to cover that node. If it cannot, it will then *expand* it, *i.e.* add some childs to it. Again this may fail if the control location this node is mapped to in the transition system has no out-going transitions. Finally the explore step ensures that the node is not an error node. If it is the case then we jump to the *check* step. If not, a new node is chosen and the exploration goes on. Nodes to explore are chosen using a DFS traversal of the tree.

Initially the unwinding consists of a single node 1 mapped to the initial location '.'. As we said we first try to cover it, but cannot since it is the only node in the tree. So the node is expanded: only the INIT? transition can be fired from '.'. A single child is then added to 1, leading to a new node 2 mapped to location 's' (cf. fig. 2(a)). Exploration goes on from 2. Again this node cannot be covered: it is not mapped to the same location as the only other one in the tree, so it is expanded. This adds five new nodes to the tree: 3 to 7, mapped respectively to '!', and the last four to 's'. Since the unwinding is expanded through a DFS traversal, node 3 is picked for expansion. Node 3 has no outgoing transition so it cannot be expanded. But it is mapped to the error location: we then stop the exploration to check the spuriousness of the path from 1 to 3: $1 \xrightarrow{\text{INIT?}} 2 \xrightarrow{\text{ERROR?}} 3$.

Assume we use the strongest post-condition interpolant computation method. Recall that all nodes are initially labeled by the full variable domain, denoted $\mathbb{Z}^d$ where d is the number of variables in the system. We then check the trace spuriousness computing $I_0 = \mathbb{Z}^d$ (node 1 label), $I_1 = \mathsf{post}_{\text{INIT?}}(\mathbb{Z}^d)$, and $I_2 = \mathsf{post}_{\text{ERROR?}}(I_1) = \varnothing$ since clearly INIT $\cap$ ERROR $= \varnothing$. Then $I_0 = \mathbb{Z}^d, I_1 = \text{INIT}, I_2 = \varnothing$ is an interpolant for the path and the path is spurious. We then go to the refinement step.

Refinement consists in updating labels of each node on the path. We intersect its current label with the corresponding set in the interpolant: node 1 is relabeled by $\mathbb{Z}^d \cap I_0 = \mathbb{Z}^d$, node 2 by $\mathbb{Z}^d \cap$ INIT, and node 3 by $\varnothing$. After labels have been updated, the algorithm tries to cover the nodes on the path. None can be covered for now. Refinement is terminated, we go back to the system exploration.

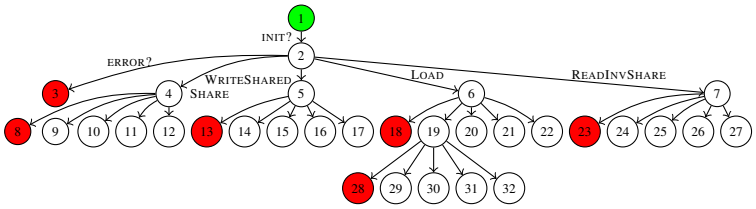
The DFS continues on its traversal of the tree. The next node to be visited is 4. It is expanded and nodes 8 to 12 are added to the tree. The DFS then visits node 8 and first tries to cover it. The only other node mapped to location '!' in the tree is 4 which is now labeled by $\varnothing$. Obviously 4 cannot cover 8. It cannot be expanded either but it is an error node. We then need to check that the path from 1 to 8 is not an actual error path in the system. We again jump to the check step.

The path $1 \xrightarrow{\text{INIT?}} 2 \xrightarrow{\text{SHARE}} 4 \xrightarrow{\text{ERROR?}} 8$ is again spurious since we can compute an interpolant for it: $I_0 = \mathbb{Z}^d$, $I_1 = \text{INIT}$, $I_2 = \varnothing$, $I_3 = \varnothing$ because INIT does not satisfy the guard of transition SHARE. Nodes labels are updated and we try to cover nodes on the path. This time we can cover 8 by 3, 4 by 2. We then go back to exploration.

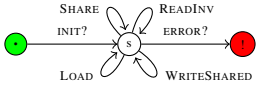
The same happens with node 5 because INIT does not satisfy the guard of transtion WRITESHARED. Node 5 is covered by node 2 and node 13 by node 3.

C.2 How the DFS may degenerate

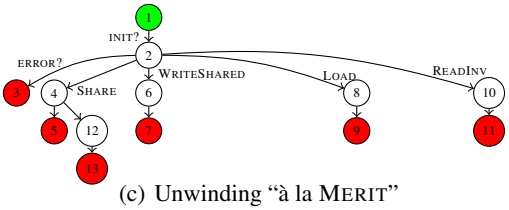
We have reached the point where we can explain why we claim that the unwinding strategy that consist in adding all children of node at once can be harmful for the algorithm termination. The exploration DFS will now visit node 6. This node is mapped to location 's' in the system and is labeled by $\mathbb{Z}^d$. Candidates to cover 6 are nodes 4 and 5. Both have been refined and their labels no longer entails $\mathbb{Z}^d$. Node 6 is then expanded. Nodes 18 to 22 are thus added to the unwinding. The branch $1 \xrightarrow{\text{INIT?}} 2 \xrightarrow{\text{LOAD}} 6 \xrightarrow{\text{ERROR?}} 18$ is refined, and then we visit node 19. The problem here is that node 19 can be covered by node 7 which has not been refined yet. Thus we will not visit 19 subtree yet. For the same reason, neither will we visit nodes 20 to 22 sub-trees. So no nodes below 6 will be expanded for now. We then have to visit node 7. Again we first refine the branch $1 \xrightarrow{\text{INIT?}} 2 \xrightarrow{\text{READINV}} 7 \xrightarrow{\text{ERROR?}} 23$. This refinement uncovers nodes 19 to 22. But when we visit nodes 24 to 27 we can then cover them by any of the nodes we just uncovered. So, again, none of the nodes below 7 are expanded. Because each time we try to expand a node mapped to location 's', we can find a node to cover it, the unwinding tree cannot grow in depth. Instead it grows in width and the more it grows the more nodes labeled with $\mathbb{Z}^d$ we have. The number of nodes grows exponentially, but we gain very little information on the actual system behaviour, since very few refinements are made. This phenomenon slows down the algorithm convergence terribly, eventually preventing its termination.



(a) Original unwinding method



(b) The Transition-system



(c) Unwinding "à la MERIT"

Fig. 2. Verification of the MESI model

C.3 MERIT unwinding strategy

We now describe how MERIT unwinds the control-structure and how it helps preventing the DFS traversal from degenerating into a BFS. The unwinding tree built by the algorithm is (partially) depicted on fig. 2(c).

Like in the previous case we start with a single node in the tree: 1 mapped to the '.' location. Node 1 is expanded which adds node 2 mapped to 's'. In turn 2 is expanded, but this time we only add a single child. Assuming children are added in the same order as before, the new node 3, is mapped to location '!'. Like before, 3 cannot be covered and since no transition leaves the '!' location, it cannot be expanded either. But it is an error node and a first refinement is made on the branch $1 \xrightarrow{\text{INIT?}} 2 \xrightarrow{\text{ERROR?}} 3$.

Assume we use weakest pre-conditions interpolant computation method this time. The branch labels are updated as follows: node 3 is relabeled with $I_2 = \varnothing$, 2 with $I_1 = \widetilde{\mathsf{pre}}_{\text{ERROR?}}(\overline{\varnothing}) = \overline{\text{ERROR}}$ and 1 with $I_0 = \mathbb{Z}^d$.

Note another difference with the previous strategy: the DFS traversal responsible for expanding and visiting all nodes in the tree has now ran out of node. And there is no uncovered leaf. But obviously the tree labels do not provide an invariant for the safety of the system. An additional condition to the *safe* termination of the algorithm is that all nodes not labeled by $\varnothing$ nor covered *must* have as many children as the location they are mapped to have outgoing transitions.

According to these conditions, the algorithm has not finished yet. We have to choose a node to start a new DFS to further explore the system. MERIT implements different heuristics for that. We will present here the one that gives the best experimental results. The node is selected by a BFS traversal of the tree, that selects the highest possible node in the tree. The idea behind this heuristic is to explore a completely different sub-tree in the system. Following this heuristic, node 2 is selected to be the root of the new DFS.

So 2 is expanded a second time and it is added a second child: 4, mapped to location 's'. Then 4 is expanded with its first child 5 which is mapped to the error location. The branch $1 \xrightarrow{\text{INIT?}} 2 \xrightarrow{\text{SHARE}} 4 \xrightarrow{\text{ERROR?}} 5$ is then refined. Node 5 is labeled by $\varnothing$, node 4 by $\overline{\text{ERROR}}$, node 2 is further refined so that it does not entails $\overline{\text{ERROR}}$ and 2 cannot cover 4, and finally 1 remains the same.

Then 2 is expanded a third time and we add node 6 and 7 to the unwinding tree. Visiting 7 triggers the refinement of the branch $1 \xrightarrow{\text{INIT?}} 2 \xrightarrow{\text{WRITESHARED}} 6 \xrightarrow{\text{ERROR?}} 7$. Node 7 is labeled by $\varnothing$, node 6 by $\overline{\text{ERROR}}$ and can thus be covered by 4. Similarly nodes 8 and 10 are covered by 4.

Finally, nodes 4 get expanded a second time and nodes 11 and 12 are added to the tree and the branch $1 \xrightarrow{\text{INIT?}} 2 \xrightarrow{\text{SHARE}} 4 \xrightarrow{\text{SHARE}} 12 \xrightarrow{\text{ERROR?}} 13$ has to be refined. This time node 12 and 13 labels will be refined as well as node 4. Update of this later label will uncover nodes 6, 8, and 8. This is not a surprise: we may have eliminated covers by nodes that have not been refined, but we cannot prevent uncovers when a node is further refined.

Transitions firing order An interesting fact one can note here, is that if we had chosen to fire transitions in an alternate order, for instance READINV, LOAD, WRITESHARED, SHARE, we may have made different cover/uncover. The impact of the order in which transitions are fired can be important: with the MESI model, using the above order

and the weakest pre-conditions interpolant computation technique, MERIT requires 85 refinements to conclude to the system safety. With the order we used to explain the unwinding process, only 25 refinements are required ! If tools like FAST allow the user to specify an order in which fire the transition, MERIT does not. It fires them in the reverse order they are read in the input model file. So far, we did not make any attempt to find ways to compute a "best order" in which fire the transitions. Moreover, since we have fully automatic techniques in mind, the benchmark results we present do not use any experimentally chosen "best order".

C.4 Influence of the interpolant computation technique

We now look at the influence of the interpolant computation technique. On the same example, we compare the weakest pre-condition and the so called cut-post computation techniques. With this later technique, when an interpolant for a branch $\pi = s_0, \ldots, s_n$ is computed, we first search for the node s_k where π becomes unsatisfiable. It proceeds by computing the weakest pre-conditions for each nodes starting from the error node s_n up towards s_0 (cf. fig. 3). Once s_k is found, the interpolant for π is the sequence of sets $I_i = \mathbb{Z}^d$ for $i = 0, \ldots, k$; $I_i = \mathsf{post}_{\tau_i}(I_{i-1})$ for $i > k$.

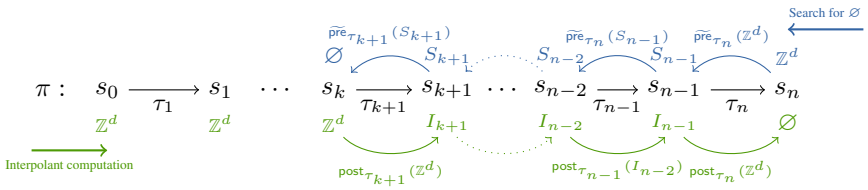


Fig. 3. Computation of a *cut-post* interpolant for a path $s_0, \ldots, s_n$

As one can see, with this interpolant computation method we obtain very precise information on the system on nodes in the bottom of the unwinding tree, while retaining a very coarse abstraction on nodes near the unwinding tree root. It is then easier to cover nodes deep in the tree with nodes that are near the root. Note however that the interpolant computed with the cut-post technique are not as precise as the ones computed with the post technique, even on the lower part of the tree. This is because we start our computation from $\mathbb{Z}^d$, not the actual set of configurations of the system: we use $\mathsf{post}_{\tau_{k+1}}(\mathbb{Z}^d)$ instead of $\mathsf{post}_{\tau_{k+1}}(C_k)$ where C_k is the exact set of configurations reachable after transitions $\tau_1 \ldots \tau_k$ have been fired. In practice, this interpolant computation technique proved to be the most efficient. But there are still some cases where these interpolant are not precise enough, as shown by the last line in the Table 1: on the ILLINOIS cache coherence protocol, this technique fails.