Lazy Constrained Monotonic Abstraction

Zeinab Ganjei(✉), Ahmed Rezine, Petru Eles, and Zebo Peng

Linköping University, Linköping, Sweden
zeinab.ganjei@liu.se

Abstract. We introduce Lazy Constrained Monotonic Abstraction (lazy CMA for short) for lazily and soundly exploring well structured abstractions of infinite state non-monotonic systems. CMA makes use of infinite state and well structured abstractions by forcing monotonicity wrt. refinable orderings. The new orderings can be refined based on obtained false positives in a CEGAR like fashion. This allows for the verification of systems that are not monotonic and are hence inherently beyond the reach of classical analysis based on the theory of well structured systems. In this paper, we consistently improve on the existing approach by localizing refinements and by avoiding to trash the explored state space each time a refinement step is required for the ordering. To this end, we adapt ideas from classical lazy predicate abstraction and explain how we address the fact that the number of control points (i.e., minimal elements to be visited) is a priori unbounded. This is unlike the case of plain lazy abstraction which relies on the fact that the number of control locations is finite. We propose several heuristics and report on our experiments using our open source prototype. We consider both backward and forward explorations on non-monotonic systems automatically derived from concurrent programs. Intuitively, the approach could be regarded as using refinable upward closure operators as localized widening operators for an a priori arbitrary number of control points.

Keywords: Constrained monotonic abstraction · Lazy exploration · Well structured systems · Safety properties · Counter machines reachability

1 Introduction

Well structured transition systems (WSTS:s for short) are maybe everywhere [17], but not all transition systems are well structured [3,18]. Problems such as state reachability (e.g., safety) have been shown to be decidable for WSTS:s [2,17]. This led to the development of algorithms that could check safety for systems ranging from lossy channels and Petri Nets to concurrent programs and broadcast protocols [19,23,25]. Many interesting examples of systems, including list manipulating programs [9], cache protocols [13] and mutex algorithms [1] are "almost" well structured in the sense that they would have been well structured

In part supported by the 12.04 CENIIT project.

B. Jobstmann and K.R.M. Leino (Eds.): VMCAI 2016, LNCS 9583, pp. 147–165, 2016.
DOI: 10.1007/978-3-662-49122-5_7

if it was not for a number of transitions that violate the required assumptions. We build on the framework of Constrained Monotonic Abstraction (CMA for short) where we derive well structured abstractions for infinite state systems that are "almost" well structured.

To simplify, a WSTS comes with a well quasi ordering (wqo[1] for short) on the set of configurations. A key property of such systems is *monotonicity*: i.e., if a smaller configuration can fire a transition and get to some configuration c, then any configuration that is larger (wrt. the wqo) can also get to some configuration that is larger than c. In other words, larger configurations simulate smaller ones. Added to some assumptions on the effectivity of natural operations such as computing minimal elements and images of upward closed sets of configurations, it is possible to show the existence of sound and complete algorithms for checking the reachability of upward closed sets of configurations (i.e., coverability).

Systems where only some transitions are non monotonic can be approximated using WSTS:s by adding abstract transitions to restore monotonicity (monotonic abstraction). The resulting abstraction is also infinite state, and reachability of upward closed sets there is decidable. However, the obtained abstractions may fail to enforce invariants that are crucial for establishing unreachability of bad configurations in the original system. For instance, we explain in our recent work [18] how we automatically account for the number of processes synchronizing with (dynamic) barriers when establishing or refuting local (e.g., assertions) and global (e.g., deadlock freedom) properties of programs manipulating arbitrary many processes. Crucial invariants of such systems enforce an inherently non-monotonic behavior (e.g., a barrier transition that is enabled on a configuration is disabled if more processes are considered in a larger configuration).

Checking safety for such non-monotonic systems is not guaranteed to terminate without abstraction. Plain monotonic abstraction [1,20] makes use of sets that are upward closed wrt. natural orderings as a sound symbolic representation. As stated earlier, this ensures termination if the used preorder is a wqo [2]. Of course, this comes at the price of possible false positives. In [3], we adapted existing counter example guided abstraction refinement (CEGAR) ideas to refine the ordering in plain monotonic abstraction. The preorder is strengthened by only relating configurations that happen to be in the same equivalence class, as defined by Craig interpolants obtained from the false positives. The new preorder is also a wqo, and hence, termination is again ensured. As implemented, the predicates are applied on all generated minimal elements to separate upward closed sets and the exploration has to restart from scratch each time a new refinement predicate is encountered.

We address these inefficiencies by adopting a lazy approach. Like in lazy predicate abstraction [21], we strive to localize the application of the refinement predicates and to reuse the explored state space. However, a major difference with plain lazy predicate abstraction is that the number of "control locations"

[1] A reflexive and transitive binary relation $\preceq$ over some set A is a preorder. It is said to be a wqo over A if in any infinite sequence $a_1, a_2, \ldots$ of elements of A, there exist $1 \leq i < j$ such that $a_i \preceq a_j$.

(i.e., the locations to which subsets of the refinement predicates are mapped) is a priori unbounded (as opposed to the number of program locations of a non-parameterized system). We propose in this paper three heuristics that can be applied both in backward and in forward (something plain monotonic abstraction is incapable of). All three heuristics adopt a backtracking mechanism to reuse, as much as possible, the state space that has been explored so far. Schematically, the first heuristic (*point-based*) associates refinement predicates to minimal elements. The second heuristic (*order-based*) associates the refinement predicates to preorder related minimal elements. The third heuristic (*descendants-based*) uses for the child the preserved predicates of the parent. We describe in details the different approaches and state the soundness and termination of each refinement step. In addition, we experimentally compare the heuristics against each other and against the eager approach on our open source tool https://gitlab.ida.liu.se/apv/zaama.

Related Work. Coverability of non-monotonic systems is undecidable in general. Tests for zero are one source of non-monotonicy. The work in [8] introduces a methodology for checking coverability by using an extended Karp-Miller acceleration for the case of Vector Addition Systems (VAS:s for short) with at most one test for zero. Our approach is more general and tackles coverability and reachability for counter machines with arbitrary tests.

Verification methods can be lazy in different ways. For instance, Craig interpolants obtained from program runs can be directly used as abstractions [26], or abstraction predicates can be lazily associated to program locations [21]. Such techniques are now well established [5,10,27]. Unlike these approaches, we address lazy exploration for transition systems with "infinite control". Existing WSTS based abstraction approaches do not allow for the possibility to refine the used ordering [23,25], cannot model transfers for the local variables [16], or make use of accelerations without termination guarantees [7]. For example, in [23] the authors leverage on the combination of an exact forward reachability and of an aggressive backward approximation, while in [25], the explicit construction of a Petri Net is avoided.

The work in [24] gives a generalization of the IC3 algorithm and tries to build inductive invariants for well-structured transition systems. It is unclear how to adapt it to the kind of non-monotonic systems that we work with.

We believe the approach proposed here can be combined with such techniques. To the best of our knowledge, there is no previous work that considered making lazy the preorder refinement of a WSTS abstraction.

Outline. We start in Sect. 2 with some preliminaries. We then formalize targeted systems and properties in Sect. 3. We describe the adopted symbolic representation in Sect. 4 and go through a motivating example in Sect. 5. This is followed by a description of the eager and lazy procedures in Sect. 6. We finally report on our experiments in Sect. 7 and conclude in Sect. 8.

2 Preliminaries

We write $\mathbb{N}$ and $\mathbb{Z}$ to respectively mean the sets of natural and integer values. We let $\mathbb{B} = \{\texttt{tt}, \texttt{ff}\}$ be the set of boolean values. Assume in the following a set X of integer variables. We write $\xi(X)$ to mean the set of *arithmetic expressions* over X. An arithmetic expression e in $\xi(X)$ is either an integer constant k, an integer variable x in X, or the sum or difference of two arithmetic expressions. We write $e(X)$ to emphasize that only variables in X are allowed to appear in e. We write $\texttt{atomsOf}(X)$ to mean the set of *atoms* over the variables X. An atom α is either a boolean $\texttt{tt}$ or $\texttt{ff}$ or an inequality $e \sim e'$ of two arithmetic expressions; where $\sim \in \{<, \leq, \geq, >\}$. We write A to mean a set of atoms. Observe that the negation of an atom can be expressed as an atom. We often write ψ to mean a conjunction of atoms, or *conjunct* for short, and use Ψ to mean a set of conjuncts. We use $\Pi(\xi(X))$ to mean arbitrary conjunctions and disjunctions of atoms over X. We can rewrite any presburger predicate over X in negated normal form and replace the negated inequalities with the corresponding atoms to obtain an equivalent predicate π in $\Pi(\xi(X))$. We write $\texttt{atomsOf}(\pi)$ to mean the set of atoms participating in π.

A mapping $\mathbb{m} : U \to V$ associates an element in V to each element in U. We write $\mathbb{m} : U \nrightarrow V$ to mean a partial mapping from U to V. We write $\texttt{dom}(\mathbb{m})$ and $\texttt{img}(\mathbb{m})$ to respectively mean the domain and the image of $\mathbb{m}$ and use $\epsilon_U : U \nrightarrow V$ for the mapping with an empty domain. We often write a partial mapping $\mathbb{m} : U \nrightarrow V$ as the set $\{u \leftarrow \mathbb{m}(u) |\ u \in \texttt{dom}(\mathbb{m})\}$ and write $\mathbb{m} \cup \mathbb{m}'$ to mean the union of two mappings $\mathbb{m}$ and $\mathbb{m}'$ with disjoint domains. Given a partial mapping $\mathbb{x} : X \nrightarrow \xi(X)$, we write $\nu_{\mathbb{x}}(e)$ to mean the substitution in e of X variables by their respective $\mathbb{x}$ images and the natural evaluation of the result. As usual, $\nu_{\mathbb{x}}(e)$ is a well defined integer value each time $\mathbb{x}$ is a total mapping to $\mathbb{Z}$. This is generalized to (sets of) atoms, conjuncts and predicates.

We let $\mathbb{X}$ (resp. $\mathbb{X}_{\geq 0}$) be the set of all total mappings $X \to \mathbb{Z}$ (resp. $X \to \mathbb{N}$). We write $\mathbb{0}_X$ for the total mapping $X \to \{0\}$. The denotation of a conjunct ψ over $\mathbb{X}$ (resp. $\mathbb{X}_{\geq 0}$), written $[\![\psi]\!]_{\mathbb{X}}$ (resp. $[\![\psi]\!]_{\mathbb{X}_{\geq 0}}$), is the set of all total mappings $\mathbb{x}$ in $\mathbb{X}$ (resp. in $\mathbb{X}_{\geq 0}$) s.t. $\nu_{\mathbb{x}}(\psi)$ evaluates to $\texttt{tt}$. We generalize to sets of atoms or conjuncts by taking the union of the individual denotations. Let $\trianglelefteq$ be the preorder over $\mathbb{X}_{\geq 0}$ defined by $\mathbb{x} \trianglelefteq \mathbb{x}'$ iff $\mathbb{x}(x) \leq \mathbb{x}'(x)$ for each $x \in X$. Given a predicate π in $\Pi(\xi(X))$, we say that a set $M \subseteq [\![\psi]\!]_{\mathbb{X}_{\geq 0}}$ is minimal for ψ if: (i) $\mathbb{x} \not\trianglelefteq \mathbb{x}'$ for any pair of different $\mathbb{x}, \mathbb{x}' \in M$, and (ii) for any $\mathbb{x}' \in [\![\psi]\!]_{\mathbb{X}_{\geq 0}}$, there is an $\mathbb{x} \in M$ s.t. $\mathbb{x} \trianglelefteq \mathbb{x}'$. We recall the following facts from Linear Programming and [22].

Lemma 1. *For a finite set of natural variables X, the preorder $\trianglelefteq$ is a partial well quasi ordering. In addition, we can compute a finite and unique minimal set (written $\boldsymbol{min}_{\trianglelefteq}(\pi)$) for any predicate π in $\Pi(\xi(X))$.*

3 The State Reachability Problem

In this section, we motivate and formally define the reachability problem.

```
    shared :
    bool read := *;

    process :
t0. pc0 → pc0 : spawn
t1. pc0 → pc1 : read := tt
.....
//do some work before the barrier
.....
t2. pc1 → pc2 : read := ff
t3. pc2 → pc3 : barrier()
t4. pc3 → pc4 : assert(!read)
```

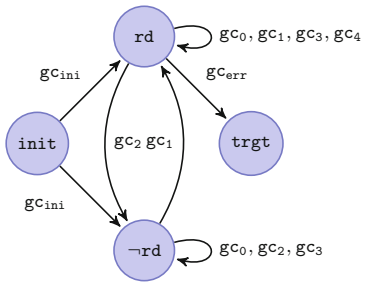

Fig. 1. The counter machine to the right captures the behaviour of the concurrent program to the left. It makes use of one counter per program location. It involves the following guarded commands: $\mathtt{gc_{ini}} ::= (c_0, c_1, c_2, c_3, c_4 := 1, 0, 0, 0, 0)$, $\mathtt{gc_0} ::= (c_0 \geq 1 \Rightarrow c_0 := c_0 + 1)$, $\mathtt{gc_1} ::= (c_0 \geq 1 \Rightarrow c_0, c_1 := c_0 - 1, c_1 + 1)$, $\mathtt{gc_2} ::= (c_1 \geq 1 \Rightarrow c_1, c_2 := c_1 - 1, c_2 + 1)$, $\mathtt{gc_3} ::= ((c_2 \geq 1 \wedge c_0 + c_1 = 0) \Rightarrow (c_2, c_3 := c_2 - 1, c_3 + 1))$, $\mathtt{gc_4} ::= (c_3 \geq 1 \Rightarrow c_3, c_4 := c_3 - 1, c_4 + 1)$, and $\mathtt{gc_{err}} : (c_3 \geq 1)$. The resulting system is not well structured because of the zero test in $\mathtt{gc_3}$.

An Example. Consider the multi-threaded program to the left of Fig. 1 where only a single thread starts executing the program. A thread can spawn arbitrarily many concurrent threads with $\mathtt{t_0}$. Assume all threads asynchronously run the same program. Each thread can then set the shared flag `read` to `tt`, and perform some reading followed by resetting `read` to `ff`. All threads wait at the barrier. Obviously, `read` should be `ff` after the barrier since all threads that reached $\mathtt{pc_3}$ must have executed $\mathtt{t_2}$. The assertion at $\mathtt{pc_3}$ should therefore hold no matter how many threads are spawned. Capturing the barrier behaviour is crucial for establishing the non-violation of the assertion. The barrier behaviour is inherently non monotonic (adding more threads does not keep the barrier open). Our recent work [18] on combining different abstraction techniques can automatically generate non-monotonic counter machines such as the one to the right of Fig. 1. For this case, the assertion in the concurrent program is violated iff the target state is reachable in the counter machine. We explain briefly in the following how such counter machines are generated.

Our tool Pacman [18], takes as input multi-threaded programs similar to the one to left of Fig. 1. It automatically performs predicate, counter and monotonic abstractions on them and generates counter machines that overapproximate the behaviour of the original program. It then tries to solve the reachability problem for those machines.

Given a multi-threaded program, Pacman starts by generating concurrent boolean programs by performing predicate abstraction and incrementally improving it in a CEGAR loop [14]. This results in a boolean multi-threaded program that has the same control flow graph as the original program, but consists of only boolean variables. To the obtained boolean program, Pacman applies counter abstraction to generate a counter machine. Intuitively, each counter in the machine is associated to each local state valuation of a thread (that consists

in the location and the valuation of the local variables of the thread). Each state in the machine is also associated to a valuation of shared variables. An extra state is reserved for the target. The statements of the boolean program are then translated as transitions in the counter machine.

For instance, in Fig. 1, counters $\mathtt{c_i}$, for $\mathtt{i} : 0 \leq \mathtt{i} \leq 4$, correspond respectively to the number of threads in program locations $\mathtt{pc_i}$ (the threads have no local variables here). Similarly, each transition $\mathtt{gc_i}$ is associated to each $\mathtt{t_i}$. Moreover, there are two additional transitions $\mathtt{gc_{ini}}$ and $\mathtt{gc_{err}}$ to model transitions involving initial or target states.

Note that the original multi-threaded program has non-monotonic invariants. For instance, transitions such as barriers, or any transition that tests variables representing the number of threads satisfying some property do not stay enabled if we add more threads. At the same time, the boolean concurrent programs generated above are inherently monotonic. This corresponds to a loss of precision. Thus, proving correctness of those programs whose correctness depends on respecting the non-monotonic behaviour (e.g., the one enforced by a barrier) can become impossible. As a remedy to this fact, PACMAN automatically strengthens counter machine transitions by enforcing barrier invariants or by deriving new invariants (e.g., using an instrumented thread modular analysis) to regain some of the precision. This proved to help in verifying several challenging benchmarks. For example, consider the transition $\mathtt{t_3}$ in the program to the left of Fig. 1. At the moment a thread crosses the barrier first, there should be no thread before location $\mathtt{pc_2}$. This fact holds afterwards and forbids that a thread sets the flag `read` when some thread is checking the assertion. The transition $\mathtt{gc_3}$ is its corresponding transition in the strengthened counter machine. To ease the presentation of the example, $\mathtt{gc_3}$ is strengthened with the guard $(\mathtt{c_0} + \mathtt{c_1} = 0)$. (Observe that this is a simplification to ease the presentation; we can more faithfully capture the barrier by combining the test with a global flag.)

Counter machines. A counter machine is a tuple $(Q, C, A, \Delta, q_{init}, q_{trgt})$ where Q is a finite set of states, C and A are two distinct sets of counters (i.e., variables ranging over $\mathbb{N}$), Δ is a finite set of transitions and q_{init} and q_{trgt} are two states in Q. A transition δ in Δ is of the form $(q, (grd \Rightarrow cmd), q')$ where $src(\delta) = q$ is the source state, $dst(\delta) = q'$ is the destination state and $\mathtt{gc}(\delta) = (grd \Rightarrow cmd)$ is the guarded command. A guard grd is a predicate in $\Pi(\xi(A \cup C))$ and a command cmd is a multiple assignment $c_1, \ldots, c_n := e_1, \ldots, e_n$ that involves $e_1, \ldots e_n$ in $\xi(A \cup C)$ and pairwise different $c_1, \ldots c_n$ in C.

Semantics. A *configuration* is a pair $\theta = (q, \mathbb{c})$ with the state $\mathtt{st}(\theta) = q$ in Q and the valuation $\mathtt{val}(\theta) = \mathbb{c}$ in $\mathbb{C}_{\geq 0} : C \to \mathbb{N}$. We let Θ be the set of configurations. We write $\theta \preceq \theta'$ to mean $\mathtt{st}(\theta) = \mathtt{st}(\theta')$ and $\mathtt{val}(\theta) \trianglelefteq \mathtt{val}(\theta')$ (see Sect. 2). The relation $\preceq$ is a partial order over Θ. In fact, the pair $(\Theta, \preceq)$ is a partial well quasi ordering [22]. Given two configurations $(q, \mathbb{c})$ and $(q', \mathbb{c}')$ and a transition $\delta \in \Delta$ with $q = src(\delta)$, $q' = dst(\delta)$ and $\mathtt{gc}(\delta) = (grd \Rightarrow (c_1, \ldots, c_n := e_1, \ldots, e_n))$, we write $(q, \mathbb{c}) \xrightarrow{\delta} (q', \mathbb{c}')$ to mean that there exists an $\mathbb{a} \in \mathbb{A}_{\geq 0}$ s.t. $\nu_{\mathbb{a} \cup \mathbb{c}}(grd)$ evaluates to $\mathtt{tt}$ and $\mathbb{c}'(c_i) = \nu_{\mathbb{c} \cup \mathbb{a}}(e_i)$ for each c_i in C. The auxiliary variables

allow us to capture transfers (needed by predicate abstraction of concurrent programs). For instance, $(c_0 \geq 1 \wedge c_0 = a_0 \wedge c_1 = a_1 \wedge a_0 + a_1 = a_2 + a_3) \Rightarrow (c_0, c_1, c_2, c_3 := 0, 0, a_2 + c_2, a_3 + c_3)$ captures situations where at least a thread is at $\mathtt{pc_0}$ and all threads at $\mathtt{pc_0}$ and $\mathtt{pc_1}$ move to $\mathtt{pc_2}$ and $\mathtt{pc_3}$. A run ρ is a sequence $\theta_0\theta_1\cdots\theta_n$. We say that it *covers* the state $\mathtt{st}(\theta_n)$. The run is *feasible* if $\mathtt{st}(\theta_0) = q_{init}$ and $\theta_{i-1} \xrightarrow{\delta_i} \theta_i$ for $i : 1 \leq i \leq n$. We write $\rightarrow$ for $\cup_{\delta\in\Delta} \xrightarrow{\delta}$.

Reachability. The reachability problem for a machine $(Q, C, A, \Delta, q_{init}, q_{trgt})$ is to decide whether it has a feasible run that covers q_{trgt}.

4 Symbolic Representation

Assume a machine $(Q, C, A, \Delta, q_{init}, q_{trgt})$. We introduce (operations on) symbolic representations used in our reachability procedures in Sect. 6.

Boxes. A *box* $\mathbb{b}$ over a set A of atoms is a partial mapping from A to booleans $\mathbb{B}$. Intuitively, a box corresponds to a bitvector denoting an equivalence class in classical predicate abstraction. We use it to constrain the upward closure step. The predicate $\psi_{\mathbb{b}}$ of a box $\mathbb{b}$ is $\wedge_{\alpha\in\mathtt{dom}(\mathbb{b})}((\mathbb{b}(\alpha) \wedge \alpha) \vee (\neg\mathbb{b}(\alpha) \wedge \neg\alpha))$ ($\mathtt{tt}$ is used for the empty box). Observe that this predicate is indeed a conjunct for any fixed box $\mathbb{b}$ and that $[\![\psi_{\mathbb{b}}]\!]$ does not need to be finite. We write $\mathbb{b}_{\mathtt{tt}}$ for the box of the $\mathtt{tt}$ conjunct. We will say that a box $\mathbb{b}$ is weaker than (or is entailed by) a box $\mathbb{b}'$ if $\psi_{\mathbb{b}'} \Rightarrow \psi_{\mathbb{b}}$ is valid. We abuse notation and write $\mathbb{b} \Leftarrow \mathbb{b}'$. Observe this is equivalent to $[\![\psi_{\mathbb{b}}]\!] \subseteq [\![\psi_{\mathbb{b}'}]\!]$.

Constraints. A *constraint* over a set A of atoms is a triplet $\phi = (q, \mathbb{c}, \mathbb{b})$ where $\mathtt{st}(\phi) \in Q$ is the state of the constraint, $\mathtt{val}(\phi) = \mathbb{c}$ is its minimal valuation, and $\mathtt{box}(\phi) = \mathbb{b}$ over A is its box. We use Φ to mean a set of constraints. A constraint $(q, \mathbb{c}, \mathbb{b})$ is well formed if $\nu_{\mathbb{c}}(\psi_{\mathbb{b}})$ holds. We only consider well formed constraints. We write $\mathtt{clo}(\mathbb{c}, \mathbb{b})$ to mean the conjunct $(\wedge_{c\in C}(c \geq \mathbb{c}(c)) \wedge \psi_{\mathbb{b}})$. Intuitively, $\mathtt{clo}(\mathbb{c}, \mathbb{b})$ denotes those valuations that are both "in the box" and in the $\trianglelefteq$-upward closure of $\mathbb{c}$. We let $[\![(q, \mathbb{c}, \mathbb{b})]\!]$ be the set $\{(q, \mathbb{c}')|\ \mathbb{c}' \in [\![\mathtt{clo}(\mathbb{c}, \mathbb{b})]\!]\}$. This set contains at least $(q, \mathbb{c})$ by well formedness. Given two constraints $(q, \mathbb{c}, \mathbb{b})$ and $(q', \mathbb{c}', \mathbb{b}')$, we write $(q, \mathbb{c}, \mathbb{b}) \sqsubseteq (q', \mathbb{c}', \mathbb{b}')$ to mean that: (i) $q = q'$, and (ii) $\mathbb{c} \trianglelefteq \mathbb{c}'$, and (iii) $\mathbb{b} \Leftarrow \mathbb{b}'$. Observe that $\phi \sqsubseteq \phi'$ implies $[\![\phi']\!] \subseteq [\![\phi]\!]$. A subset Φ of a set of constraints Φ' is minimal if: (i) $\phi_1 \not\sqsubseteq \phi_2$ for any pair of different constraints $\phi_1, \phi_2 \in \Phi$, and (ii) for any $\phi' \in \Phi'$, there is a $\phi \in \Phi$ s.t. $\phi \sqsubseteq \phi'$.

Lemma 2. *For a finite set of atoms A over C, the ordering $\sqsubseteq$ is a well quasi ordering over the set of well formed constraints over A. In addition, we can compute, for any set Φ of constraints, a finite $\sqsubseteq$-minimal subset $\mathit{min}_{\sqsubseteq}(\Phi)$.*

Image Computations. Assume a conjunct ψ over C and a guarded command $gc = (grd \Rightarrow cmd)$ for some $\delta \in \Delta$. Recall that grd is in $\Pi(\xi(C \cup A))$ and that cmd is of the form $c_1, \ldots, c_n := e_1, \ldots, e_n$ where, for each $i : 1 \leq i \leq n$, c_i is in C and e_i is also in $\xi(C \cup A)$. We let L' be the set of primed versions of all variables

appearing in the left hand side of *cmd*. We write $\texttt{pre}_{gc}(\psi)$ to mean a set of conjuncts whose disjunction is equivalent to $(\exists A \cup L'.(\wedge_{1\leq i\leq n}(c'_i = e_i) \wedge grd \wedge \psi[\{c \leftarrow c' |\ c' \in L'\}]))$. We also write $\texttt{post}_{gc}(\psi)$ to mean a set of conjuncts whose disjunction is equivalent to $(\exists A \cup C.(\wedge_{1\leq i\leq n}(c'_i = e_i) \wedge grd \wedge \psi))[\{c' \leftarrow c |\ c \in C\}]$. We naturally extend $\texttt{pre}_{gc}(\psi)$ and $\texttt{post}_{gc}(\psi)$ to sets of conjuncts.

Lemma 3. *Assume $\delta \in \Delta$ and conjuncts Ψ. We can compute $\texttt{pre}_{gc(\delta)}(\Psi)$ and $\texttt{post}_{gc(\delta)}(\Psi)$ s.t. $[\![\texttt{pre}_{gc(\delta)}(\Psi)]\!]$ (resp. $[\![\texttt{post}_{gc(\delta)}(\Psi)]\!]$) equals $\{\mathbb{c} |\ (src(\delta), \mathbb{c}) \xrightarrow{\delta} (dst(\delta), \mathbb{c}')$ with $\mathbb{c}' \in [\![\Psi]\!]\}$ (resp. $\{\mathbb{c}' |\ (src(\delta), \mathbb{c}) \xrightarrow{\delta} (dst(\delta), \mathbb{c}')$ with $\mathbb{c} \in [\![\Psi]\!]\}$).*

Grounded Constraints and Symbolic Sets. A *grounded constraint* is a pair $\gamma = ((q, \mathbb{c}, \mathbb{b}), \psi)$ that consists of a constraint $\texttt{cstrOf}(\gamma) = (q, \mathbb{c}, \mathbb{b})$ and a conjunct $\texttt{groundOf}(\gamma) = \psi$. It is well formed if: $(q, \mathbb{c}, \mathbb{b})$ is well formed, $\psi \Rightarrow \texttt{clo}(\mathbb{c}, \mathbb{b})$ is valid, and $\mathbb{c} \in [\![\psi]\!]$. We only manipulate well formed grounded constraints. Intuitively, the ground ψ in $((q, \mathbb{c}, \mathbb{b}), \psi)$ represents the "non-approximated" part of the $\trianglelefteq$-upward closure of $\mathbb{c}$. This information will be needed for refining the preorder during the analysis. We abuse notation and write $\texttt{cstrOf}(\Gamma)$, resp. $\texttt{groundOf}(\Gamma)$, to mean the set of constraints, resp. grounds, of a set Γ of grounded constraints. A trace σ of length n is a sequence starting with a grounded constraint followed by n transitions and grounded constraints. We say that two traces $(\phi_0, \psi_0)\cdot\delta_1\cdot(\phi_1, \psi_1)\cdots\delta_n\cdot(\phi_n, \psi_n)$ and $(\phi'_0, \psi'_0)\cdot\delta'_1\cdot(\phi'_1, \psi'_1)\cdots\delta'_{n'}\cdot(\phi'_{n'}, \psi'_{n'})$ are equivalent if: (i) $n = n'$, and (ii) δ_i is the same as δ'_i for each $i : 1 \leq i \leq n$, and (iii) $\phi_i \sqsubseteq \phi'_i$, $\phi'_i \sqsubseteq \phi_i$ and $\psi_i \Leftrightarrow \psi'_i$ for each $i : 0 \leq i \leq n$. A symbolic set is a set of pairs of grounded constraints and traces. Given a symbolic set T, we also use $\texttt{cstrOf}(T)$ to mean all constraints ϕ appearing in some $((\phi, \psi), \sigma)$ in T. Recall that we can compute a set $\texttt{min}_{\sqsubseteq}(\texttt{cstrOf}(T))$ of $\sqsubseteq$-minimal constraints for $\texttt{cstrOf}(T)$.

5 An Illustrating Example

We use the example introduced in Sect. 3 to give an intuition of the lazy heuristics described in this paper. A more detailed description follows in Sect. 6.

Plain monotonic abstraction proceeds backwards while systematically closing upwards wrt. the natural ordering $\preceq$ on Θ. The trace depicted in Fig. 2 is a generated false positive. In this description, for $i : 0 \leq i \leq 7$, we write $\gamma_i = (\phi_i, \psi_i)$ to mean the grounded constraint with the grounded constraint ψ_i and the constraint $\phi_i = (q_i, \mathbb{c}_i, \mathbb{b}_i)$. Intuitively, the grounded constraint represents "exact" valuations while the constraint captures over-approximations that are of the form $(q_i, \mathbb{c})$ where $\mathbb{c}_i \trianglelefteq \mathbb{c}$ and $\mathbb{c}$ satisfies $\psi_{\mathbb{b}_i}$. The computation starts from the grounded constraint $\gamma_7 = ((\texttt{trgt}, \mathbb{c}_7, \mathbb{b}_{\texttt{tt}}), \psi_7)$ where ψ_7 is $\wedge_{c\in C}(c \geq 0)$ (always implicit). For γ_7, the exact and the over-approximated parts coincide.

The trace then computes $\psi_6 = (c_3 \geq 1)$ which captures the valuations of the predecessors of $(\texttt{trgt}, \mathbb{c}_7, \mathbb{b}_{\texttt{tt}})$ wrt. $(\texttt{rd}, \texttt{gc}_{\texttt{err}}, \texttt{trgt})$. This set happens to be upward closed and there is no need for approximation, hence $\gamma_6 = ((\texttt{rd}, \mathbb{c}_6, \mathbb{b}_{\texttt{tt}}), \psi_6)$. Valuations of the exact predecessors of $(\texttt{rd}, \mathbb{c}_6, \mathbb{b}_{\texttt{tt}})$ wrt.

$(\texttt{rd}, \texttt{gc}_3, \texttt{rd})$ are captured with the conjunct $\psi_5 = (c_0 = c_1 = 0 \wedge c_2 \geq 1)$. These are approximated with the conjunct $(c_0 \geq 0 \wedge c_1 \geq 0 \wedge c_2 \geq 1)$. Continuing to compute the predecessors and closing upwards leads to the constraint ϕ_0 which involves the initial state `init`. The trace is reported as a possible reachability witness. It is well known [4] that upward closed sets are not preserved by non-monotonic transitions (such as those involving $\texttt{gc}_3$ in Fig. 1). At the same time, maintaining an exact analysis makes guaranteeing termination impossible.

Following the trace in forward from the left, it turns out that the upward closure that resulted in γ_5 is the one that made the spurious trace possible. Indeed, it is its approximation that allowed the counter c_1 to be non zero. This new value for c_1 is the one that allowed the machine to execute $(\neg\texttt{rd}, \texttt{gc}_1, \texttt{rd})$ in backward from ϕ_5, making reaching the initial state possible. The constraint ϕ_5 is the pivot constraint of the trace. Constrained monotonic abstraction (CMA) proposes to refine the used ordering by strengthening it with a relevant predicate. In this case, $c_1 \leq 0$ is used for strengthening, but in general (the atoms of) any predicate in $\Pi(\xi(C))$ that separates the exact predecessors from the reachable part of the upward closure would do.

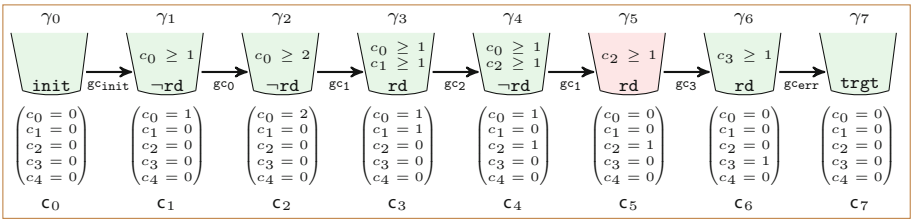

Fig. 2. A spurious trace generated by monotonic abstraction. The γ_5 constraint introduces the first over-approximation that makes the spurious trace possible. The configuration $(\texttt{rd}, \mathbb{c}_5)$ is the pivot configuration of the spurious trace.

Eager CMA. Introduced in [3]. The exploration is restarted from scratch and $(c_1 \leq 0)$ is used to systematically partition all exact predecessors. The upward closure is constrained to not alter the refinement predicate. All generated valuations are therefore approximated with the stronger ordering. Localizing refinement can make possible both reusing a potentially large part of the explored state space and applying the (slower) refinement to a smaller number of sets.

Lazy CMA. When backtracking, we only eliminate those constraints that were obtained as descendants of a constraint that needs to be refined. We refer to this constraint as the pivot constraint, and to its minimal configuration as the pivot configuration. In fact, we identify three localization heuristics:

- *point-based-lazy.* We map the refinement predicates to the pivot configurations. Later in the exploration, when we hit a new pivot configuration, we constrain wrt. those predicates that were already mapped to it.
- *order-based-lazy.* The point-based approach may be too localized as there is an infinite number of pivot configurations. For instance, a similar trace can

continue, after $(\mathtt{rd}, c_2 = 1)$, with $\mathtt{gc}_1$ and get to the minimal configuration sending c_2 to 2. This one is different from the mapped pivot configuration, and hence we need to introduce a new pivot configuration with the same predicate $c_0 \leq 0$. This approach considers the predicates of all larger or smaller pivot configurations. The idea being that, if the predicate was important for the mapped pivot configuration, then it must have been to separate it from a reachable upward closed part, and hence it may be relevant.

– *descendants-based-lazy.* In addition to associating refinement predicates to pivot configurations as in the point-based approach, this heuristic leverages on the fact that predicates may remain relevant for a sequence of transitions. Here we compare the exact predecessors with the predicates used to constrain the upward closure of the parent. If those predicates still hold for the predecessors, then we maintain them when closing upwards. This heuristic bears similarity to forward propagation of clauses in IC3 [24], as in the IC3 algorithm the clauses are propagated in the trace from a preceding formula to the succeeding one if they still hold.

6 State Reachability Checking

We describe in this section four different forward CMA variants (*eager*, *point-based-lazy*, *order-based-lazy* and *descendants-based-lazy*). The four procedures can also be applied in backwards (as described in the experiments of Sect. 7). The four variants use grounded constraints as symbolic representations for possibly infinite numbers of machine configurations. The symbolic representation is refined using atoms obtained using a counterexample guided refinement scheme. The difference between the four variants lays in the way discovered predicates (in fact atoms for simplifying the presentation) are associated to the new symbolic representations and in the way backtracking is carried out. We start by introducing the basic "partition" procedure.

Input: a state q, a conjunct ψ and a finite set of atoms A
Output: a well formed set of grounded constraints
1 $\Gamma := \emptyset$;
2 **foreach** ($total$ $\mathbb{b} : A \to \mathbb{B}$) **do**
3 | **foreach** ($\mathbb{c} \in \mathit{min}_{\trianglelefteq}(\psi \wedge \psi_{\mathbb{b}})$) **do** $\Gamma := \Gamma \cup ((q, \mathbb{c}, \mathbb{b}), \psi \wedge \mathtt{clo}(\mathbb{c}, \mathbb{b}))$;
4 **return** Γ;

Procedure partition(q, ψ, A) is common to all variants.

Partition. "partition(q, ψ, A)" partitions ψ according to all atoms in A. Each obtained conjunct is further decomposed according to its $\trianglelefteq$-minimal valuations. Conjuncts are then used to build a well formed grounded constraint $((q, \mathbb{c}, \mathbb{b}), \psi')$ where $\mathbb{b}$ is a box over A. Observe that the disjunction of the grounds of the obtained grounded constraints is equivalent to ψ. Soundness is stated in Lemma 4.

Lemma 4. *Assume a finite set A of atoms. For any conjunct ψ, it is the case that* $[\![(q, \psi)]\!] = \{(q, \mathbb{c}) \mid \mathbb{c} \in [\![\psi']\!]_{\geq 0}$ *for each* $\psi' \in \mathit{groundOf}(\text{partition})(q, \psi, A))\} \subseteq [\![\mathit{cstrOf}(\text{partition})(q, \psi, A))]\!]$.

```
   Input: a machine M = (Q, C, A, Δ, q_init, q_trgt)
   Output: A feasible run covering q_trgt or the value unreachable
1  if q_init = q_trgt then return (q_init, 0_C);
2  S, Γ := ∅, partition(q_init, ∧_{c∈C}(c ≥ 0)), ∅);
3  foreach (γ ∈ Γ) do S := S ∪ {(γ, γ)} ;
4  return explore(M, S, S, ∅, ε_Θ);
```

Procedure checkReachability(M) is the common entry point for all variants.

Eager CMA, like the other variants, starts by passing a description of the machine to the "checkReachability" procedure. It returns a feasible run covering q_{trgt}, or states that there are no such runs. The procedure returns directly (line 1) if initial and target states coincide. It then calls "partition" to obtain a set of well formed grounded constraints that together capture all initial configurations. These are passed to the "explore" procedure.

Explore. "explore(M, `work`, `store`, `sleep`, $\mathbb{f}$)" results in a working list process that maintains three symbolic sets `work`, `store` and `sleep`. The last is only relevant for the lazy variants. The partial mapping $\mathbb{f} : \Theta \nrightarrow$ `atomsOf`(C) encapsulates all refinement predicates discovered so far and is therefore empty when the procedure is called from "checkReachability". Intuitively, $\mathbb{f}(\theta)$ associates to the pivot configuration θ those predicates that helped eliminate a false positive when θ was the minimal configuration of the constraint that made the false positive possible. We will explain how $\mathbb{f}$ is updated when introducing the procedure "simulate". The symbolic set `work` is used for the grounded constraints that are yet to be visited (i.e., for which the successors are still to be computed and approximated). The `store` set is used for both those grounded constraints that have been visited and for those in *working*. The `sleep` set corresponds to those constraints that might have to be visited but for which there is an $\sqsubseteq$-equivalent representative in `store`. In case a backtracking eliminates the representative in `store`, the corresponding grounded constraint in `sleep` has to be reconsidered. This is explained in the "backtrack" procedure of the lazy variants.

```
   Input: A machine description M = (Q, C, A, Δ, q_init, q_trgt), three symbolic sets work, store
          and sleep, and a partial mapping 𝕗 : Θ ↛ atomsOf(C)
   Output: A feasible run covering q_trgt or the value unreachable
1  while there exists ((φ, ψ), σ) in work with φ ∈ min_⊑(cstrOf(store)) do
2  |  remove ((φ, ψ), σ) from work;
3  |  (q, ℂ, 𝔹) := φ ;
4  |  if q = q_trgt then
5  |  |  return simulate(M, work, store, sleep, 𝕗, σ);
6  |  foreach δ = (q, gc, q') in Δ do
7  |  |  foreach ψ_p ∈ post_gc(clo(ℂ, 𝔹)) do
8  |  |  |  foreach (φ', ψ') in decompose(q', ψ_p, 𝕗, 𝔹) do
9  |  |  |  |  σ' := σ · δ · (φ', ψ');
10 |  |  |  |  if there is ((φ_e, ψ_e), σ_e) in store s.t. φ_e is ⊑-equivalent to φ' then
11 |  |  |  |  |  if σ_e and σ' are not equivalent then
12 |  |  |  |  |  |  add ((φ', ψ'), σ') to sleep;
13 |  |  |  |  else add ((φ', ψ'), σ') to both store and work ;
14 return unreachable;
```

Procedure explore(M, `work`, `store`, `sleep`, $\mathbb{f}$) is common to all variants.

The procedure picks a pair $((\phi, \psi), \sigma)$ from `work` and $\texttt{min}_{\sqsubseteq}(\texttt{cstrOf}(\texttt{store}))$. If the initial state is reached, it calls procedure "simulate" to check the associated trace and to backtrack if needed (lines 4–5). Otherwise, we start by iterating through all transitions δ in Δ and compute an exact representation of the predecessors of the constraint. The call "decompose$(q, \psi_p, \mathbb{f}, \mathbb{b})$" boils down, for the eager variant, to a call to "partition$(q, \psi_p, \texttt{img}(\mathbb{f}))$". The obtained grounded constraints are used to update the `store`, `work` and `sleep` symbolic sets.

If there was no pair picked at line 1, then we have finished the exploration and return unreachable. In fact, pairs are never removed from `store` if no target states are encountered at line 4. In addition, two pairs with $\sqsubseteq$-equivalent constraints cannot be added to `work` (lines 10–13). For this reason, executing the first line an infinite number of times without calling procedure "simulate" would result in an infinite sequence of constraints that would violate Lemma 2.

Input: machine M, symbolic sets `work`, `store` and `sleep`, a mapping $\mathbb{f} : \Theta \nrightarrow \texttt{atomsOf}(C)$ and a trace $\sigma = (\phi_0, \psi_0) \cdot \delta_1 \cdots \delta_n \cdot (\phi_n, \psi_n)$ with $n \geq 1$ and $q_0 = q_{init}$ and $q_n = q_{trgt}$;
Output: A feasible run covering q_{trgt} or the value `unreachable`

1 $\Psi_n := \{\psi_n\}$;
2 **for** $i \leftarrow (n-1)$ **to** 0 **do**
3 $\quad \Psi'_i := \texttt{pre}_{\texttt{gc}(\delta_{i+1})}(\Psi_{i+1})$;
4 $\quad \Psi_i := \{(\psi_i \wedge \psi'_i) |\ \psi'_i \in \Psi'_i$ and $(\psi_i \wedge \psi'_i)$ is sat$\}$;
5 $\quad$ **if** Ψ_i *is empty* **then**
6 $\quad\quad \mathbb{f}((\texttt{st}(\phi_i), \texttt{val}(\phi_i)))\cup := \{\alpha |\ \alpha \in \texttt{atomsOf}(\pi)$ with $\pi \in \textbf{ITP}(\{\psi_i\}, \Psi'_i)\}$;
7 $\quad\quad$ **return** backtrack$(M, \texttt{work}, \texttt{store}, \texttt{sleep}, \mathbb{f}, \sigma, i)$;
8 **return** a run starting at $(q_{init}, \mathbb{c})$ for some $\mathbb{c} \in \Psi_0$ and following till q_{trgt};

Procedure simulate$(M, \texttt{work}, \texttt{store}, \texttt{sleep}, \mathbb{f}, \sigma)$ is common to all variants.

Simulate. This procedure checks feasibility of a trace σ from q_{init} to q_{trgt}. The procedure incrementally builds a sequence of sets of conjuncts $\Psi_n, \ldots, \Psi_0$ where each Ψ_i intuitively denotes the valuations that are backwards reachable from q_{trgt} after k steps of σ (starting from $k = 0$), and are still denoted by $\texttt{clo}(\mathbb{c}_{(n-k)}, \mathbb{b}_{(n-k)})$. The idea is to systematically intersect (a representation of) the successors of step k with the grounded constraint that gave raise to the constraint at step $k + 1$. If the procedure finds a satisfiable Ψ_0, then a run can be generated by construction. Such a run is then returned at line 8. Otherwise, there must have been a step where the "exact" set of conjuncts does not intersect the conjunct representing the exact part that gave raise to the corresponding constraint. In other words, the trace could be eliminated by strengthening the over-approximation at line 7 of the "explore" procedure. In this case, (at line 6 of the "simulate" procedure), new refinement atoms are identified using an off-the-shelf interpolation procedure for QF_LIA (Quantifier Free Linear Arithmetic). This information will be used differently by the eager and lazy variants when calling their respective "backtrack" procedures.

Input: a machine M, sets `work` and `store` and mapping $\mathbb{f} : \Theta \nrightarrow \texttt{atomsOf}(C)$;
Output: A feasible run covering q_{trgt} or the value `unreachable`

1 `store`, `work` $:= \emptyset, \emptyset$;
2 $\Gamma :=$ partition$(q_{init}, \wedge_{c \in C}(c \geq 0), \texttt{img}(\mathbb{f}))$;
3 **foreach** (ϕ, ψ) *in* Γ **do**
4 $\quad$ S $:=$ S $\cup \{((\phi, \psi), (\phi, \psi))\}$;
5 **return** explore$(M, \texttt{S}, \texttt{S}, \emptyset, \mathbb{f})$;

Procedure backtrack$(M, \texttt{work}, \texttt{store}, _, \mathbb{f}, _, _)$ this is the eager variant.

Eager backtracking throws away the explored state space (line 1) and restarts the computation from scratch using the new refinement atoms captured in $\mathbb{f}$.

Lazy Backtracking. Intuitively, all three lazy approaches reuse the part of the explored state space that is not affected by the new refinements. This is done by restarting the exploration from new sets `work` and `store` that are obtained after pruning away the pivot constraint identified by the argument i passed by "simulate" together with all its descendants (identified in lines 1–6). One important aspect is that grounded constraints that have not been added to `store` at line 11 of the "explore" procedure may have been discarded for the wrong reason (i.e., there was an $\sqsubseteq$-equivalent constraint that needs to be pruned away now). This would jeopardize soundness. For this reason we maintain the `sleep` set for tracking the discarded grounded constraints that have to be put back to `work` and `store` if the constraint that blocked them is pruned away (see lines 4–6). The refined pivot is added to the new sets `work` and `store` (lines 10–13). Lines 7–9 are only used by the descendants-based approach which takes into account the box of the parent.

Input: symbolic sets `work`, `store` and `sleep`; a mapping $\mathbb{f} : \Theta \nrightarrow$ `atomsOf`(C), a trace $\sigma = (\phi_0, \psi_0) \cdot \delta_0 \cdots (\phi_n, \psi_n)$ with $n \geq 1$ and $\mathtt{st}(\phi_0) = q_{init}$ and $\mathtt{st}(\phi_n) = q_{trgt}$, and a natural $i : 0 \leq i < n$;
Output: A feasible run covering q_{trgt} or the value `unreachable`

1 **foreach** $((\phi, \psi), \tau) \in$ `store` *st.* $(\phi_0, \psi_0) \cdot \delta_0 \cdots (\phi_i, \psi_i)$ *is equivalent to a prefix of* τ **do**
2 remove, if present, $((\phi, \psi), \tau)$ from `work`, `store` and `sleep`;
3 **for** $j \leftarrow i$ *to* n **do**
4 **if** *there is still a* $((\phi', \psi'), \tau')$ *in* `sleep` *with* ϕ' *is* $\sqsubseteq$*-equivalent to* ϕ_j **then**
5 remove $((\phi', \psi'), \tau')$ from `sleep`;
6 add $((\phi', \psi'), \tau')$ to both `work` and `store`;
7 **if** $i \geq 1$ **then**
8 $\mathbb{b}_p := \mathbb{b}_{i-1}$
9 **else** $\mathbb{b}_p := \mathbb{b}_{\mathtt{tt}}$;
10 **foreach** $(\phi', \psi') \in$ decompose$(q_i, \psi_i, \mathbb{f}, \mathbb{b}_p)$ **do**
11 **let** $\sigma' := (\phi_0, \psi_0) \cdot \delta_1 \cdots (\phi_{i-1}, \psi_{i-1}) \cdot \delta_i \cdot (\phi', \psi')$;
12 **if** *there is some* $((\phi_e, \psi_e), \sigma_e)$ *in* `store` *st.* ϕ_e *is* $\sqsubseteq$*-equivalent to* ϕ' **then**
13 **if** σ_e *and* σ' *are not equivalent* **then**
14 add $((\phi', \psi'), \sigma')$ to `sleep`;
15 **else** add $((\phi', \psi'), \sigma')$ to both `store` and `work` ;
16 **return** explore(M,`work`,`store`,`sleep`, $\mathbb{f}$);

Procedure backtrack(M, `work`, `store`, `sleep`, $\mathbb{f}, \sigma, i$) common to all lazy variants.

The main difference between the lazy variants is in the way their respective "decompose" procedures associate refinement atoms to "exact" conjuncts.

Point-based. This variant is the one that "localizes" most the refinement. Each time an obtained grounded conjunct is considered for approximation, it checks whether its minimal valuation has already been associated to some refinement atoms. If it is the case, it passes them when calling the "partition" procedure.

Input: a state q, a conjunct ψ and a partial mapping $\mathbb{f} : \Theta \nrightarrow$ `atomsOf`(C)
Output: a well formed set of grounded constraints
1 $A := \emptyset$;
2 **foreach** $(\theta \in \mathit{dom}(\mathbb{f})$ *with* $\mathit{val}(\theta) \in \mathit{min}_{\trianglelefteq}(\psi))$ **do** $A := A \cup \mathbb{f}(\theta)$;
3 **return** partition(q, ψ, A)

Procedure decompose$(q, \psi, \mathbb{f}, -)$ of the point-based-lazy variant.

Order-based. This variant "localizes" less than the point-based variant. Each time an obtained "exact" conjunct is considered for approximation, it checks whether its minimal valuation is $\trianglelefteq$-related to an already mapped valuation. The union of all corresponding atoms is passed to the "partition" procedure.

Input: a state q, a conjunct ψ and a mapping $\mathbb{f} : \Theta \nrightarrow \texttt{atomsOf}(C)$
Output: a well formed set of grounded constraints
1 **let** $A := \emptyset$;
2 **foreach** $(\theta \in \mathit{dom}(\mathbb{f}))$ **do**
3 | **foreach** $(\mathbb{c}' \in \mathit{min}_{\trianglelefteq}(\psi))$ **do**
4 | | **if** $((\mathbb{c}' \trianglelefteq \mathit{val}(\theta))$ *or* $(\mathit{val}(\theta) \trianglelefteq \mathbb{c}'))$ **then**
5 | | | $A := A \cup \mathbb{f}(\theta)$;
6 | | | **break** ;
7 **return** partition(q, ψ, A)

Procedure decompose$(q, \psi, \mathbb{f}, _)$ of the order-based variant.

Descendants-based. This variant "localizes" less than the point-based variant, but is incomparable with the order-based one. The idea is to keep those refinement atoms that were used for the parent constraint, and that are still weaker than the current conjunct that is to be approximated.

Input: a state q, a conjunct ψ, a box $\mathbb{b}$ and a mapping $\mathbb{f} : \Theta \rightarrow \texttt{atomsOf}(C)$
Output: a well formed set of grounded constraints
1 **let** $A := \emptyset$;
2 **foreach** $(\theta \in \mathit{dom}(\mathbb{f})$ *with* $\mathit{val}(\theta) \in \mathit{min}_{\trianglelefteq}(\psi))$ **do** $A := A \cup \mathbb{f}(\theta)$;
3 **foreach** $\alpha \in \mathit{dom}(\mathbb{b})$ **do**
4 | **if** $(\mathbb{b}(\alpha) \wedge (\psi \Rightarrow \alpha))$ *or* $(\neg\mathbb{b}(\alpha) \wedge (\psi \Rightarrow \neg\alpha))$ **then**
5 | | $A := A \cup \{\alpha\}$;
6 **return** partition(q, ψ, A)

Procedure decompose$(q, \psi, \mathbb{f}, \mathbb{b})$ of the descendants-based variant.

Finally, we state the soundness of our four exploration variants. The proof is by observing that `store` always represents, at the i^{th} iteration of the loop of procedure "explore", an over-approximation of the machine configurations obtained after i steps. Combined with Lemmas 2 and 3 and by well quasi ordering of $\sqsubseteq$ on the set of constraints for a finite number of refinement atoms.

Theorem 1. *All four exploration variants are sound. In addition, each call to procedure "checkReachability" eventually terminates if only a finite number of calls to procedure "simulate" are executed.*

Proof. Sketch. Let $\texttt{work}_k$, $\texttt{store}_k$ and $\texttt{sleep}_k$ be the sets `work`, `store` and `sleep` obtained at line 1 at the k^{th} iteration of the loop in procedure "explore". We can show the following propositions by induction on k (see the appendix for the details):

(a) $[\![\texttt{store}_k]\!]$ does not intersect $(q_{trgt}, \mathbb{c})$ for any valuation $\mathbb{c}$
(b) $[\![\texttt{store}_k]\!]$ intersects $(q_{init}, \mathbb{c})$ for every valuation $\mathbb{c}$
(c) $[\![\texttt{work}_k \cup \texttt{sleep}_k]\!]$ is a subset of $[\![\texttt{store}_k]\!]$
(d) for each element $((\phi, \psi), \sigma)$ of $\texttt{store}_k$ such that $((\phi, \psi), \sigma) \notin \texttt{work}_k$ and $\phi \in \texttt{min}_{\sqsubseteq}(\texttt{cstrOf}(\texttt{store}_k))$ and for each transition $\delta = (q, gc, q') \in \Delta$, the configurations in $\{(q', \mathbb{c}') |\ \mathbb{c}' \in [\![\texttt{post}_{gc}(\texttt{clo}(\texttt{val}(\phi), \texttt{box}(\phi)))]\!]\}$ are also in $[\![\texttt{store}_k]\!]$

Soundness. Suppose the algorithm returns `unreachable`. Then at some iteration, there is no element $((\phi,\psi),\sigma)$ in `work` s.t. $\phi \in \mathtt{min}_{\sqsubseteq}(\mathtt{cstrOf}(\mathtt{store}))$. Combined with propositions (b), (c) and (d), we have that $[\![\mathtt{store}]\!]$ is a fixpoint that is an overapproximation of all reachable configurations. Proposition (a) ensures that no element with state q_{trgt} exists in `store`. If the algorithm returns a trace, then the test at line 4 ensures that $\mathtt{st}(\phi_n) = q_{trgt}$ for some $((\phi_n,\psi_n),\sigma)$ and $\sigma = (\phi_0,\psi_0)\cdot\delta_1\cdots\delta_n\cdot(\phi_n,\psi_n)$ satisfies that $\mathtt{st}(\phi_0) = q_{init}$, $\mathtt{st}(\phi_n) = q_{trgt}$ and for $0 \le i < n$, $(\mathtt{st}(\phi_i),\mathtt{val}(\phi_i)) \xrightarrow{\delta_{i+1}} (\mathtt{st}(\phi_{i+1}),\mathtt{val}(\phi_{i+1}))$. This because of the form of the added tuple at line 13 of "explore".

Termination. The procedure "checkReachability" terminates if only a finite number of calls to procedure "simulate" are executed. This relies on the fact that the only source of non-termination can be the while loop in "explore" if the set $\mathtt{cstrOf}(\mathtt{work}) \cap \mathtt{min}_{\sqsubseteq}(\mathtt{cstrOf}(\mathtt{store}))$ never becomes empty. Suppose there is an infinite sequence of constraints as $\phi_0, \phi_1 \ldots$ obtained in the while loop. First, we show that $i \neq j$ implies ϕ_i is not $\sqsubseteq$-equivalent with ϕ_j for any $i, j \ge 0$. This holds because an element is added to `store` only if there is no $\sqsubseteq$-equivalent element there (line 9 of "explore"). Even if an element is moved from `sleep` to `store` and `work` by "backtrack", then it is done after removing the $\sqsubseteq$-equivalent element in `store` and `work`. Second, we show that for any $0 \le i < j$, $\phi_i \not\sqsupseteq \phi_j$. This holds because if $\phi_i \sqsubseteq \phi_j$, then ϕ_j could not be in $\mathtt{min}_{\sqsubseteq}(\mathtt{cstrOf}(\mathtt{store}))$ since ϕ_i (or an $\sqsubseteq$-equivalent constraint) is already there. Finally, since the number of calls to "backtrack" is finite, then the number of predicates being used in the boxes is also finite. Such a sequence would therefore violate Lemma 2. □

7 Experimental Results

We have implemented our techniques in our open source tool Zaama. The tool and benchmarks are available online[2]. The tool relies on the Z3 SMT solver [12] for its internal representations and operations.

The input of the prototype are counter machine encodings of boolean multi-threaded programs with broadcasts and arbitrary tests (as described in Sect. 3). We have experimented with more than eighty different counter machine reachability problems. These were obtained from our prototype tool Pacman [18] that checks local (i.e., assertion) or a global (e.g., deadlock freedom) properties in concurrent programs (some inspired from [11,15]).

Given a property to check on a concurrent program, Pacman proceeds in predicate abstraction iterations. For each set of tracked predicates, it creates a counter machine reachability problem. Combining Pacman with Zaama results in a nested CEGAR loop: an outer loop for generating counter machine reachability problems, and an inner loop for checking the resulting problems. About 45 % of the generated counter machines are not monotonic. We tested all those

[2] https://gitlab.ida.liu.se/apv/zaama.

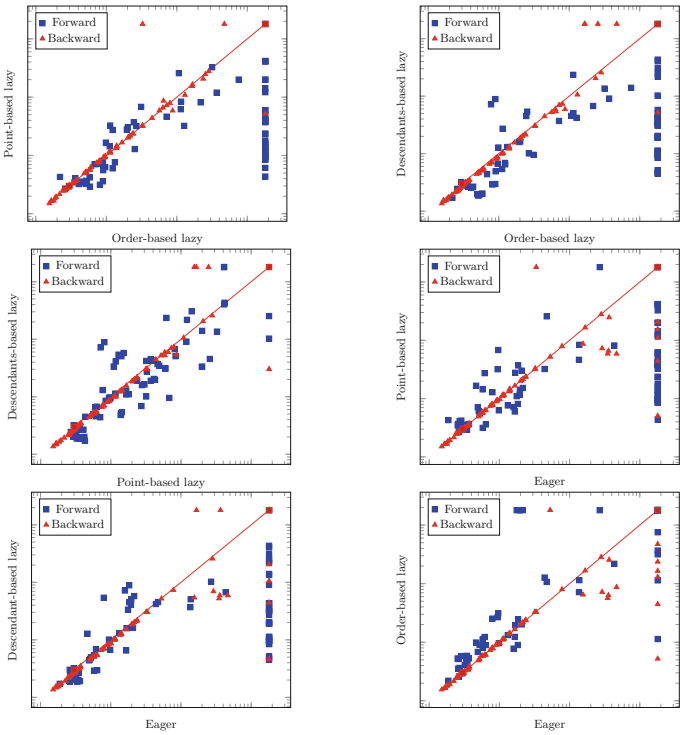

Fig. 3. Comparing eager and lazy variants on a logarithmic scale.

machines separately with ZAAMA in different settings for each benchmark and reported the execution times. Thus, the PACMAN overhead is not included in the reported times. Note that although 55 % of the examples are monotonic, they still need refinement in forward exploration.

We also tested our benchmarks with the tool BREACH introduced in [23]. BREACH cannot take non-monotonic inputs and is inherently incapable of solving reachability problems for such systems which are the main target of this paper. Thus, we could apply it only to the monotonic benchmarks; for which, the runtime of BREACH was less than 5 seconds in each. We consider this to be an encouraging result as we are working on adapting BREACH to non-monotonic systems. The challenge is to have an under-approximation search engine for such systems and we are investigating possibilities to develop our own engine or to use acceleration tools such as FASTer [6].

We have chosen a time-out of 30 min for each of the variants: eager, point-based, order-based and descendants-based, both in forward and in backward. We have conducted our experiments on an Intel Core i7 2.93 GHz processor with 8GB of memory. We report on our results in Fig. 3 where we consider, for each setting, each lazy pair in addition to the pairs consisting in the eager and each lazy.

The forward explorations turned out to be faster than the corresponding backward ones in about 25 % of the examples. We expected the forward exploration to be slower as it needs several refinement steps because it starts from the initial configurations which are typically much more constrained than the target configurations. We considered the forward exploration because it offers more possibilities to test the effect of localizing the refinement in problems that typically require more refinement steps in forward. Indeed, the figures show that the times of the different variants coincide more often in backward than in forward, and overall, there has been many more time-outs in forward than in backward.

Furthermore, the lazy variants were able to conclude on most of the reachability problems, in fact each of the reachability problems has been solved by at least one of the lazy variants (except for one problem in backward), when the eager variant timed out on several of them. This is an encouraging result that confirms the advantages of localizing refinement. There are some cases where the eager variant did better than all lazy ones. These correspond to cases where localization required more refinement efforts to reach a conclusion.

We also observe that the order-based approach times out in about half the forward seraches, while the point-based only times out in two cases. This goes against the initial intuition that larger valuations would profit from the refinement predicates of the smaller ones. One explanation could be that if the larger valuation would require the same predicate as the smaller one, then adding the predicate would result in a redundant representation that should be eliminated. It therefore seems that it does not take long for the point-based to discover this redundancy while still profiting from the localization of the refinement. Instead, the order-based uses predicates even when they are not proven to be needed resulting in finer grained symbolic elements that slow down the exploration.

It is interesting to observe that the descendants-based approach did better in forward than the point-based approach. One explanation could be that, in forward, relevant refinement interpolants sometimes correspond to weak inductive invariants that get propagated by this approach. In backwards it seems, at least for our examples, that the invariants corresponding to the "bad" configurations do not profit from this parent-child transmission.

8 Conclusion

We have introduced and discussed three different ways of localizing constrained monotonic abstraction in systems with infinite control. For this, we have targeted reachability problems for (possibly non-well structured) counter machines obtained as abstractions of concurrent programs. Our new techniques allow us to avoid systematically trashing the state space explored before encountering the false positives that necessitate the introduction of new refinement predicates. This allowed us to consistently improve on the existing eager exploration, both in forward and in backward explorations. Possible future works concern combining forward and backward approximations, using the pivot configuration to make possible the choice of interpolants that are easier to generalize and assessing the feasibility of combination with new partial order techniques.

References

1. 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)
2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: Proceedigs of the LICS 1996, 11^{th} IEEE International Symposium on Logic in Computer Science, pp. 313–321 (1996)
3. Abdulla, P.A., Chen, Y.-F., Delzanno, G., Haziza, F., Hong, C.-D., Rezine, A.: Constrained monotonic abstraction: a CEGAR for parameterized verification. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 86–101. Springer, Heidelberg (2010)
4. 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)
5. Ball, T., Rajamani, S.K.: The SLAM Project: debugging system software via static analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT, POPL 2002, pp. 1–3. ACM, New York (2002)
6. Bardin, S., Finkel, A., Leroux, J.: FASTer acceleration of counter automata in practice. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 576–590. Springer, Heidelberg (2004)
7. Bardin, S., Leroux, J., Point, G.: FAST extended release. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 63–66. Springer, Heidelberg (2006)
8. Bonnet, R., Finkel, A., Leroux, J., Zeitoun, M.: Model checking vector addition systems with one zero-test (2012). arXiv preprint arXiv:1205.4458
9. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
10. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS, pp. 570–574. Springer (2005)
11. Cogumbreiro, T., Hu, R., Martins, F., Yoshida, N.: Dynamic deadlock verification for general barrier synchronisation. In: Proceeding of the 20th ACM SIGPLAN PPoPP Symposium, pp. 150–160. ACM (2015)
12. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
13. Delzanno, G.: Automatic verification of parameterized cache coherence protocols. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 53–68. Springer, Heidelberg (2000)
14. Donaldson, A.F., Kaiser, A., Kroening, D., Tautschnig, M., Wahl, T.: Counterexample-guided abstraction refinement for symmetric concurrent programs. Formal Meth. Syst. Des. **41**(1), 25–44 (2012)
15. Downey, A.: The Little Book of SEMAPHORES (2nd Edition): The Ins and Outs of Concurrency Control and Common Mistakes. Createspace Ind, Pub (2009). http://www.greenteapress.com/semaphores/
16. Farzan, A., Kincaid, Z., Podelski, A.: Proofs that count. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 151–164. ACM, New York (2014)

17. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere!. Theor. Comput. Sci. **256**(1–2), 63–92 (2001)
18. Ganjei, Z., Rezine, A., Eles, P., Peng, Z.: Abstracting and counting synchronizing processes. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 227–244. Springer, Heidelberg (2015)
19. Geeraerts, G., Raskin, J.-F., Van Begin, L.: Expand, enlarge and check.. made efficient. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 394–407. Springer, Heidelberg (2005)
20. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 22–29. Springer, Heidelberg (2010)
21. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT, POPL 2002, pp. 58–70. ACM, New York (2002)
22. Higman, G.; Ordering by divisibility in abstract algebras. In: Proceedings of the London Mathematical Society, pp. 326–336 (1952)
23. Kaiser, A., Kroening, D., Wahl, T.: Efficient coverability analysis by proof minimization. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 500–515. Springer, Heidelberg (2012)
24. Kloos, J., Majumdar, R., Niksic, F., Piskac, R.: Incremental, inductive coverability. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 158–173. Springer, Heidelberg (2013)
25. Liu, P., Wahl, T.: Infinite-state backward exploration of boolean broadcast programs. In: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design, pp. 155–162. FMCAD Inc (2014)
26. 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)
27. Weissenbacher, G., Kroening, D., Malik, S.: Wolverine: battling bugs with interpolants. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 556–558. Springer, Heidelberg (2012)