Invariants for Finite Instances and Beyond

Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, Fatiha Zaïdi

Invariants for Finite Instances and Beyond

Sylvain Conchon*† Amit Goel‡ Sava Krstić‡ Alain Mebsout*† Fatiha Zaïdi*

*LRI, Université Paris Sud CNRS, Orsay F-91405
†INRIA Saclay – Ile-de-France, Orsay cedex, F-91893
‡Strategic CAD Labs, Intel Corporation

***Abstract*—Verification of safety properties of concurrent programs with an arbitrary numbers of processes is an old challenge. In particular, complex parameterized protocols like FLASH are still out of the scope of state-of-the-art model checkers. In this paper, we describe a new algorithm, called BRAB, that is able to automatically infer invariants strong enough to prove a protocol like FLASH. BRAB computes over-approximations of backward reachable states that are checked to be unreachable in a finite instance of the system. These approximations (candidate invariants) are then model checked together with the original safety properties. Completeness of the approach is ensured by a mechanism for backtracking on spurious traces introduced by too coarse approximations.**

I. Introduction

Nowadays, modern computing systems are often relying on multi-core or distributed architectures. Inherently, the verification of mutual exclusion or cache coherence properties for such systems is very challenging. Consider for instance that, in the Stanford FLASH multiprocessor architecture, the transition system describing the cache coherence protocol has already more than 67 million states when just four processors are in competition [9].

A standard way of verifying a transition system is to enumerate the entire state space [22], [24] (modulo reduction and compaction techniques). However, on large problems, efficient enumerative model checkers reach their limits in both time and memory consumption. For instance, Murφ fails to prove the safety of FLASH for five processes with a timeout limit of 24 hours and 20 GB of memory.

An alternative is to verify a parameterized version of the system. Model checking of such systems is an old and well studied problem [5], [12], [18]. Yet, all automatic techniques that allow properties to be verified for any number of processes do not scale very well. For instance, state-of-the-art parametric model checkers hit their limit on academic problems: most tools need several minutes to prove the safety of the parameterized protocol given by German [36], although this protocol only has 28,000 reachable states for four processes.

Some approaches are known to scale on large problems: compositional and abstraction model checking techniques [10] have been used to prove a parameterized version of FLASH [30], [38]. However, they all require human experts to provide hand-crafted invariants. Designing algorithms to find automatically good quality invariants is still a challenge and an active research area [13], [21], [28], [31], [36].

In this paper, we propose a novel algorithm that infers invariants capable of proving complex protocols. Our contributions are as follows:

- The BRAB algorithm (illustrated in Section II). It first computes a set $\mathcal{M}$ of reachable states using a forward exploration for a finite instance of the system with a *fixed number* of processes. Then, it performs a backward reachability analysis of the *parameterized* system. At each loop iteration, BRAB computes an over-approximation of backward reachable states and checks that it represents states that are not in $\mathcal{M}$. All these approximations, which can be seen as candidate invariants, are model checked together with the original safety properties. To ensure completeness, BRAB backtracks when it encounters a spurious trace introduced by a too coarse approximation.
  The strength of our method resides in two aspects. First, model checking approximations together makes it possible for the proof of an approximation to use part of the proof of another one. A second key insight is that finite instances (even small) are generally good oracles for guiding the choice of approximations as they can be seen as a concentrated knowledge of the system.
- A formalization of BRAB for a generic symbolic framework where sets of states are represented by logical formulas (Section III). We only require pre- and post-images to be computable and the decidability of backward reachability. Under these conditions, we prove soundness, completeness and termination of our algorithm. Such a generic presentation allows BRAB to be implemented in different frameworks.
- An implementation of BRAB in the framework of array-based transition systems (Section IV). This is a syntactically restricted class of parametrized transition systems with states represented as arrays indexed by an arbitrary number of processes [19]. Our implementation is available in the Cubicle model checker [14].
- A comparison of our approach with state-of-the-art parameterized and enumerative model checkers on a set of significant problems (Section V). This comparison effort demonstrates that our method is promising.

To our knowledge, Cubicle (with BRAB) is the first tool that proves automatically the safety of FLASH.

II. Invariants for Finite Instances and Beyond

We illustrate our method on a simplified version of the directory based cache coherence protocol proposed by German [36]. The protocol consists of a global directory which maintains the consistency of a shared memory between a

parameterized number of cache clients. The status of each cache i is indicated by a variable $\texttt{Cache}[i]$ which can be in one of the three states: (E)xclusive (read and write accesses), (S)hared (read access only) or (I)nvalid (no access to the memory). Clients send requests to the directory when cache misses occur: rs for a shared access (read miss), re for an exclusive access (write miss). The directory has four variables: a boolean flag Exg indicates whether a client has an exclusive access to the main memory, a boolean array Shr, such that $\texttt{Shr}[i]$ is true when a client i is granted (read or write) access to the memory, Cmd stores the current request (ϵ stands for the absence of request), and Ptr contains the emitter of the current request.

The initial states of the protocol are represented by the following logical formula

$$\forall i.\ \texttt{Cache}[i] = \mathsf{I} \ \wedge\ \neg\texttt{Shr}[i] \ \wedge\ \neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \epsilon$$

stating that the cache of each process is invalid, no access has been given and there is no request to be processed.

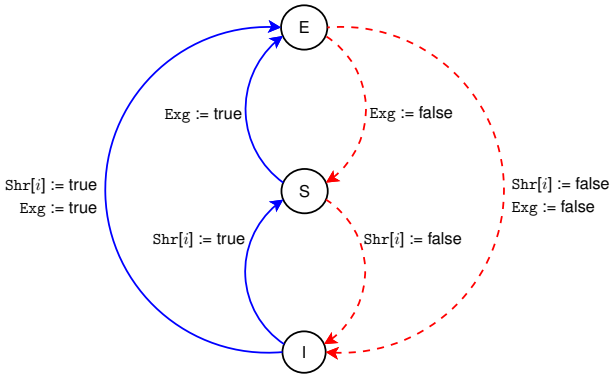

Fig. 1: State diagram of the German-*ish* protocol

We give in Figure 1 a high-level view of the evolution of a single cache. Solid arrows show the evolution of the cache following its own requests whereas, dashed arrows depict transitions resulting from a request of another client. For example, a cache moves from state I to S when a read miss occurs and the directory grants it a shared access, while recording it in the array $\texttt{Shr}[i] := \mathsf{true}$. Similarly, when a write miss occurs in another cache, the directory invalidates all clients recorded in Shr before granting an exclusive access. This has the effect of moving caches from states E or S to state I.

The formal description of the protocol is given by the transition system in Figure 2. Following notations in [36], we describe each transition by a logical formula relating the values of state variables before and after the transition. We denote by $\texttt{X}'$ the value of the variable X after the execution of the transition. For instance, transition t_1 should read as: if there exists a process i whose cache is invalid and there is no command to be processed, then update variable Ptr to i and set variable Cmd to rs.

This protocol ensures that when a cache client is in an exclusive state then no other process has (read or write) access to the memory. Proving this safety property amounts

$$t_1 : \exists i.\ \begin{array}{l}\texttt{Cache}[i] = \mathsf{I} \ \wedge\ \texttt{Cmd} = \epsilon \ \wedge \\ \texttt{Ptr}' = i \ \wedge\ \texttt{Cmd}' = \mathsf{rs}\end{array}$$

$$t_2 : \exists i.\ \begin{array}{l}\texttt{Cache}[i] \neq \mathsf{E} \ \wedge\ \texttt{Cmd} = \epsilon \ \wedge \\ \texttt{Ptr}' = i \ \wedge\ \texttt{Cmd}' = \mathsf{re}\end{array}$$

$$t_3 : \exists i.\ \begin{array}{l}\texttt{Shr}[i] \ \wedge\ \texttt{Cmd} = \mathsf{re} \ \wedge \\ \neg\texttt{Exg}' \ \wedge\ \texttt{Cache}'[i] = \mathsf{I} \ \wedge\ \neg\texttt{Shr}'[i]\end{array}$$

$$t_4 : \exists i.\ \begin{array}{l}\texttt{Shr}[i] \ \wedge\ \texttt{Cmd} = \mathsf{rs} \ \wedge\ \texttt{Exg} \ \wedge \\ \neg\texttt{Exg}' \ \wedge\ \texttt{Cache}'[i] = \mathsf{S}\end{array}$$

$$t_5 : \exists i.\ \begin{array}{l}\texttt{Ptr} = i \ \wedge\ \texttt{Cmd} = \mathsf{rs} \ \wedge\ \neg\texttt{Exg} \ \wedge \\ \texttt{Cmd}' = \epsilon \ \wedge\ \texttt{Shr}'[i] \ \wedge\ \texttt{Cache}'[i] = \mathsf{S}\end{array}$$

$$t_6 : \exists i.\ \begin{array}{l}\texttt{Ptr} = i \ \wedge\ \texttt{Cmd} = \mathsf{re} \ \wedge\ \neg\texttt{Exg} \ \wedge\ \forall j.\ \neg\texttt{Shr}[j] \\ \texttt{Cmd}' = \epsilon \ \wedge\ \texttt{Exg}' \ \wedge\ \texttt{Shr}'[i] \ \wedge\ \texttt{Cache}'[i] = \mathsf{E}\end{array}$$

Fig. 2: German-*ish* transition system

to checking that states that satisfy the following formula Θ are not reachable:

$$\Theta :\ \exists i, j.\ i \neq j \ \wedge\ \texttt{Cache}[i] = \mathsf{E} \ \wedge\ \texttt{Cache}[j] \neq \mathsf{I}$$

Finite Instance. We consider a finite instance of the protocol with two caches. We give in Figure 3 (left graph) the beginning of a forward exploration starting from the state (circled node) obtained by instantiating the initial formula with two distinct processes #1 and #2. Each edge label $t(\#_i)$ stands for the instance of a transition t with process $\#_i$.

Backward Reachability. We then run a backward reachability analysis for the parameterized system. Starting from Θ (octagon node), we iteratively compute its pre-images (circle nodes) for all transitions. Pre-images that are subsumed by already visited nodes (dotted edges) are not expanded anymore. This process ends either when a formula in a node intersects the initial formula or when there is no more pre-image to compute.

To improve this standard backward analysis, we try to prune the search space by finding over-approximations of pre-images. Since the set of possibilities is very large, we restrict the choice to formulas that represent unreachable states in the finite instance, and which are syntactic sub-formulas of pre-images.

If it succeeds to extract an appropriate candidate, the newly found approximation (rectangles connected with a bold dashed arrow) replaces the original formula. To illustrate this we show a partial graph of BRAB's execution on the right of Figure 3.

Starting from the unsafe formula $\exists i \neq j.\ \texttt{Cache}[i] = \mathsf{E} \ \wedge\ \texttt{Cache}[j] \neq \mathsf{I}$, the pre-image by transition t_5 returns the node $\exists i \neq j.\ \neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \mathsf{rs} \ \wedge\ \texttt{Ptr} = j \ \wedge\ \texttt{Cache}[i] = \mathsf{E}$. This node could be approximated by $\neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \mathsf{rs}$. But on closer inspection, $\neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \mathsf{rs}$ is already reachable on the left graph of Figure 3 as it is satisfied by a concrete state of the finite instance (double-headed arrow with $\models$) so it is undoubtedly not a good approximation. On the contrary

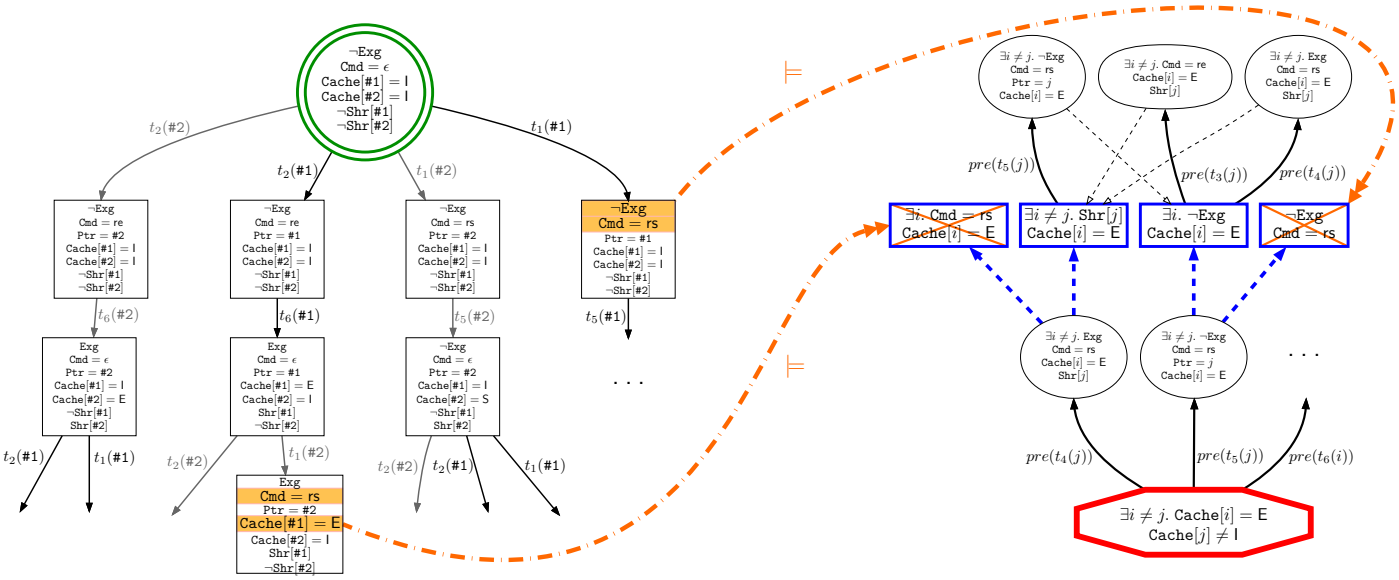

Fig. 3: BRAB on the German-*ish* protocol

no instances of $\exists i.\ \neg\texttt{Exg} \wedge \texttt{Cache}[i] = \texttt{E}$ is reachable on the finite system of Figure 3. This approximation is inserted in the backward reachability loop which continues as usual.

As we can see on the graph Figure 3, the sub-graphs of some approximations intersect. These sub-graphs depict the proof of unreachability for each approximation, this means that proofs are factorized, hence the benefit of inserting them during the main search. For example, part of the pre-image of the first approximation is subsumed by the second approximation (and *vice-versa*).

Naturally approximations can introduce spurious traces. When one is exposed, BRAB restarts from scratch the construction of the reachability graph, while remembering this approximation to avoid exploring the same spurious behaviours. For example, if we had built the finite model using only one process variable, nothing would prevent the algorithm from considering the bad approximation $\exists i.\texttt{Cmd} = \texttt{rs} \wedge \texttt{Cache}[i] = \texttt{E}$

In our example, with a finite model with two processes, no approximation introduces an error trace and the system is proved safe. As a consequence, each node in the graph is unreachable and its negation is an invariant of the system. In particular, approximations yield the following non-trivial invariants (the last one is not shown on the graph):

$$I_1: \quad \forall i, j.\ i \neq j \ \wedge\ \texttt{Cache}[i] = \texttt{E} \implies \neg\texttt{Shr}[j]$$
$$I_2: \quad \forall i.\ \texttt{Cache}[i] = \texttt{E} \implies \texttt{Exg}$$
$$I_3: \quad \forall i.\ \texttt{Cache}[i] \neq \texttt{I} \implies \texttt{Shr}[i]$$

III. Formalizing the BRAB Algorithm

A. Notations and Preliminaries

We assume the usual syntactic and semantic notions of first-order logic. In particular, we use the symbol $\models$ for the logical entailment relation between sets of formulas. For convenience, disjunctions are represented by sets of formulas.

We adopt a symbolic framework for specification of parameterized systems where states are described by a *fixed set* of state variables $\mathcal{Q}$. Each variable $x \in \mathcal{Q}$ is defined over a finite or infinite domain $\mathcal{D}_x$. This domain may be unspecified, in which case we call it a parameter of the system. We assume that in this framework (sets of) system states can be described by formulas in a decidable fragment of the first-order logic.

A parameterized system S is defined by a pair $(I, \mathcal{T})$ where I is a formula describing the initial states of the system and $\mathcal{T}$ is a set of (possibly quantified) formulas, called *transitions*, relating states of S. For a state formula φ and a transition $\tau \in \mathcal{T}$, let $pre(\tau, \varphi)$ be the formula describing the set of states from which a state satisfying φ can be reached in one τ-step. The pre-image *closure* of φ, denoted by $\textsc{Pre}^*(\varphi)$, is defined as follows

$$\begin{cases} \textsc{Pre}^0(\varphi) & \triangleq \quad \varphi \\ \textsc{Pre}^n(\varphi) & \triangleq \quad \bigcup\{pre(\tau, \psi) \mid \psi \in \textsc{Pre}^{n-1}(\varphi), \tau \in \mathcal{T}\} \\ \textsc{Pre}^*(\varphi) & \triangleq \quad \bigcup_{k \in \mathbb{N}} \textsc{Pre}^k(\varphi) \end{cases}$$

and the pre-image of a set of formulas V is defined by $\textsc{Pre}^*(V) = \bigcup_{\varphi \in V} \textsc{Pre}^*(\varphi)$. We also write $\textsc{Pre}(\varphi)$ for $\textsc{Pre}^1(\varphi)$. Similarly, we define the post-image $post(\tau, \varphi)$ of φ with respect to τ as the set of states that are reachable from φ in one step by taking the transition τ. The definition of $\textsc{Post}^*$ is given by the equations for $\textsc{Pre}^*$, with *post* in place of *pre*. For our purpose, we assume $\textsc{Pre}$ to be effectively computable and $\textsc{Post}$ to be effectively computable on finite instances.

Definition 1. *A set of formulas V is said to be* reachable *iff $\textsc{Post}^*(I) \wedge V$ is satisfiable, or equivalently, $\textsc{Pre}^*(V) \wedge I$ satisfiable. It is* unreachable *otherwise.*

Definition 2. *An* invariant *of a system is any property that holds in all reachable states of the system.*

We give a standard backward reachability algorithm for this framework, as defined by the function `BWD` in Algorithm 1. Starting with an empty set $\mathcal{V}$ of *visited nodes* (state formulas/set really) and a queue $\mathcal{Q}$ of *pending nodes* initialized

with a formula Θ, BWD iteratively computes the backward reachability graph of $\text{PRE}^*(\Theta)$. The algorithm terminates when a node fails the *safety check* (consistency with the initial condition — line 6), or when all nodes in $\mathcal{Q}$ are *subsumed* by the nodes in $\mathcal{V}$ (line 8).

The decidability of BWD is assumed to be guaranteed in the symbolic framework under consideration.

Algorithm 1: Backward Reachability Analysis

Input: a parameterized system $(I, \mathcal{T})$ and a formula Θ
Variables:
$\mathcal{V}$: visited nodes
$\mathcal{Q}$: work queue

```
1  function BWD() : begin
2      V := ∅;
3      push(Q, Θ);
4      while not_empty(Q) do
5          φ := pop(Q);
6          if (I ∧ φ sat) then
7              return unsafe
8          else if (φ ⊭ V) then
9              V := V ∪ {φ};
10             push(Q, PRE(φ));
11         end
12     end
13     return safe
14 end
```

B. The BRAB Algorithm

BRAB is defined by Algorithms 2 and 3. It implements an extended version of backward reachability that computes over-approximations during the search loop and backtracks when spurious traces are introduced by too coarse formulas.

BRAB takes as input a parameterized system $(I, \mathcal{T})$, a formula Θ, and two integers $d_{\max}$ and k. In addition to the set $\mathcal{V}$ of visited nodes and the work queue $\mathcal{Q}$, our algorithm requires three variables $\mathcal{M}$, $\mathcal{B}$ and $\mathcal{F}$, and a couple of maps Kind and From. These variables are used as follows:

- $\mathcal{M}$ is a set of reachable states for a finite instance of the system with k processes;
- $\mathcal{B}$ is a set of bad (or too coarse) over-approximations;
- $\mathcal{F}$ contains the last visited node that fails the safety check during the backward analysis;
- Kind maps formulas to values in $\{\text{Orig}, \text{Appr}\}$. It is used to differentiate formulas in $\text{PRE}^*(\Theta)$ (Orig formulas) from pre-images of over-approximations (Appr formulas);
- From associates pre-images with their original ancestor formula.

The entry point of the algorithm is the function BRAB. It starts by initializing some variables : $\mathcal{B}$ is the empty set, Θ is recorded as the initial value of $\text{PRE}^*(\Theta)$ in Kind and From, and $\mathcal{M}$ is the set $\text{FWD}(d_{\max}, k)$ of reachable states constructed by a forward exploration of the reachability graph starting in $I(\#1) \wedge \cdots \wedge I(\#k)$ and limited to depth $d_{\max}$. BRAB then enters a verification loop (line 34). It first calls the function BWDA which verifies the safety of Θ in the parameterized case by a backward reachability with approximations. If BWDA returns *safe*, so does BRAB. Otherwise, $\mathcal{F}$ contains the last formula that fails the safety check in BWDA and BRAB returns *unsafe* if $\mathcal{F}$ is a pre-image of Θ. If $\mathcal{F}$ is (a pre-image of) an approximation, then BRAB ignores this spurious result, saves the original ancestor of $\mathcal{F}$ in $\mathcal{B}$ to avoid reproducing the same trace, and continues its verification loop.

BWDA implements a reachability loop similar to Algorithm 1. It only differs at two lines. It saves in $\mathcal{F}$ the formula which fails the safety check (line 21). It also calls function Approx in place of PRE (line 25) to find over-approximations of the current node φ. The function Approx limits potential candidates to subformulas subsuming φ that are not already known to be bad approximations and which represent states that are not in $\mathcal{M}$. If it fails to find such an approximation, Approx returns the pre-image of φ. Regarding the correctness of BRAB, the set $\texttt{candidates}(\varphi)$ must be finite (implementation details are given in Section IV). If Approx finds a new approximation ψ, it is tagged with Appr in Kind and $\texttt{From}(\psi)$ is set to ψ. Otherwise, formulas in $\text{PRE}(\varphi)$ inherit the information of φ in Kind and From.

C. Correctness

The correctness of BRAB relies on the decidability of BWD (assumed in Section III-A) the following loop invariants:

(Inv1) $\mathcal{V}$ does not contain *immediately* reachable formulas, *i.e.* $\mathcal{V} \models \neg I$

(Inv2) $\text{PRE}^*(\Theta)$ is *incrementally* computed in $\mathcal{V}$ and $\mathcal{Q}$, *i.e.* $\text{PRE}^*(\Theta) \models \mathcal{V} \vee \text{PRE}^*(\mathcal{Q})$

(Inv3) if $\texttt{Kind}(\varphi) = \text{Orig}$ then $\varphi \in \text{PRE}^*(\Theta)$

Theorem 1. *If* BRAB() *returns* safe *then* Θ *is unreachable.*

Proof: When BRAB returns *safe*, the loop (line 27) terminates with $\mathcal{Q}$ empty. Now, by contradiction, suppose Θ is reachable. By definition $\text{PRE}^*(\mathcal{V}) \wedge I$ is satisfiable. Since $\mathcal{Q}$ is empty, by (Inv2) $\text{PRE}^*(\Theta) \models \mathcal{V}$ and $\mathcal{V} \wedge I$ thus satisfiable too, which contradicts the invariant (Inv1) $\mathcal{V} \models \neg I$. ■

Theorem 2. *If* BRAB() *returns* unsafe *then* Θ *is reachable*

Proof: If BRAB returns *unsafe*, then there exists a formula φ such that $\varphi \wedge I$ is satisfiable and $\texttt{Kind}(\varphi) = \text{Orig}$. By (Inv3), we conclude that $\text{PRE}^*(\Theta) \wedge I$ is satisfiable. ■

Theorem 3. BRAB() *always returns* safe *or* unsafe

Proof: Suppose the algorithm 2 does not terminate then whether:

1) BWDA does not terminate, or
2) BRAB does not terminate

(1) Since BWDA only differs from BWD by Approx, its termination is assured by the fact that candidates returns a finite set of formuals. (2) The co-domain of From is a subset of

Algorithm 2: Backward Reachability with Approximations and Backtracking (BRAB)

Input: a parameterized system $(I, \mathcal{T})$, a formula Θ to be verified, the maximal depth $d_{\max}$ for the forward exploration and the number k of processes to be considered for the finite instance of the system
Variables:
$\mathcal{V}$: visited nodes
$\mathcal{Q}$: work queue
$\mathcal{M}$: Finite model obtained by forward exploration
$\mathcal{B}$: bookkeeping of bad approximations
$\mathcal{F}$: last node visited when unsafety discovered
`Kind`: map formulas $\longmapsto$ {Orig, Appr}
`From`: map formulas $\longmapsto$ formula

```
1  function Approx(φ) : begin
2      foreach ψ in candidates(φ) do
3          if ψ ∉ B ∧ M ⊭ ψ then
4              Kind(ψ) := Appr;
5              if Kind(φ) = Orig then From(ψ) := ψ
               else From(ψ) := From(φ)  return ψ
6          end
7      end
8      foreach ψ in PRE(φ) do
9          Kind(ψ) := Kind(φ);
10         From(ψ) := From(φ)
11     end
12     return PRE(φ)
13 end
14
15 function BWDA() : begin
16     V := ∅;
17     push(Q, Θ);
18     while not_empty(Q) do
19         φ := pop(Q);
20         if (I ∧ φ sat) then
21             F := φ;
22             return unsafe
23         else if (φ ⊭ V) then
24             V := V ∪ {φ};
25             push(Q, Approx(φ))
26         end
27     end
28     return safe
29 end
30
31 function BRAB() : begin
32     B:= ∅; Kind(Θ) := Orig; From(Θ) := Θ;
33     M := FWD(d_max,k);
34     while BWDA() = unsafe do
35         if Kind(F) = Orig then return unsafe;
36         B := B ∪ {From(F)};
37     end
38     return safe
39 end
```

Algorithm 3: Finite and Depth-Limited Forward Analysis

Input: a parameterized system $(I, \mathcal{T})$
Variables:
$\mathcal{V}$: visited nodes
$\mathcal{Q}$: work queue

```
1  function FWD(d_max,k) : begin
2      V := ∅;
3      push(Q, (0, I(#1) ∧ ··· ∧ I(#k)));
4      while not_empty(Q) do
5          (d, φ) := pop(Q);
6          if (φ ∉ V and d ≤ d_max) then
7              V := V ∪ {φ};
8              N := {(d + 1, ψ) | ψ ∈ POST(φ)};
9              push (Q, N)
10         end
11     end
12     return V
13 end
```

$\mathcal{A} = \bigcup_{\varphi \in \mathcal{V}_f}$ `candidates(`φ`)` (guaranteed by line 5), where $\mathcal{V}_f$ is the final set obtained by `BWD` on Θ. Since $\mathcal{V}_f$ is finite, $\mathcal{A}$ is also finite. The condition $\psi \notin \mathcal{B}$ at line 3 guarantees that approximations added in $\mathcal{B}$ at line 36 are always distinct and in $\mathcal{A}$. In other words $\mathcal{B}$ cannot grow forever, so `BRAB` terminates. ■

Remark Notice that the correctness of BRAB does not depend on the content of $\mathcal{M}$, which thus acts as an oracle.

IV. Implementation

We have implemented BRAB in the logical framework of array-based transition systems that was proposed by Ghilardi and Ranise [19]. In this framework, states are represented by infinite arrays indexed by processes. This class is useful to model several infinite state systems and allows some topology constraints to be specified on indexes (linear ordering, multisets). Although this framework does not have all the desirable properties of Section III-A for completeness, Theorem 1 is still applicable.

In this framework, unsafe properties are cubes (conjunctions of literals existentially quantified by distinct variables). Safety properties are decidable by backward reachability as soon as a well-quasi ordering can be exhibited on models (configurations). The interested reader is referred to [19] for further details. We present here the choices we made for this implementation and its practical aspects.

In `FWD`, the construction of $\mathcal{M}$ only relies on the implementation of POST. Computing POST symbolically could be advantageous for some problems but we found out that a forward *enumerative* exploration worked best (efficiency wise) on our benchmarks. This forces us to abstract away all variables living in unbounded domains and can lead to a model where unreachable states were explored. Since the correctness does not depend on the finite model in any way, its construction can incorporate any state-of-the-art enumerative techniques or methods tailored to bug finding. For instance, its precision could be improved by adding a way of restricting

unbounded types (*e.g.* to handle infinite systems with arithmetic operations). In any case, the only significant quality of the partial model is to be able to disprove the majority of wrong approximations.

In `Approx`, to ensure termination of BRAB, we restrict `candidates(`φ`)` to a finite set of strict syntactic subformulas of φ. In Cubicle, nodes of the proof graph are cubes, seen as sets of literals, so when looking for an approximation, we successively test all its subsets starting from the coarsest ones, *i.e.* the ones that represent the largest sets of states. Choosing first the most general approximations will yield stronger invariants. We keep the first that is not disproved by $\mathcal{M}$ or the set $\mathcal{B}$. Notice this forbids us to directly approximate an approximation. In some cases, no suitable approximation can be found, and we continue the algorithm as usual. Going further than our implementation of `candidates`, for instance to consider all formula ψ such that $\psi \models \varphi$, is possible but is a complicated matter as there exists infinitely many such ψ. In addition, the framework guarantees that pre-images of cubes are computable as disjunctions of cubes.

The efficiency of BRAB relies essentially on two aspects: the choice of approximations and the content of the set $\mathcal{M}$. Indeed, choosing an approximation that is not general enough will delay the convergence of the algorithm and inserting a too general approximation will lead to unnecessary restarts. Similarly, if the exploration of the finite instance is incomplete or too imprecise, some wrong approximation will not be detected before the backward reachability exposes an error trace, resulting in a costly restart. In our implementation, we limit the negative effect of restarts trying to do them as early as possible, by finding wrong approximations sooner. For example, it is a wise choice to give a higher priority in the queue to pre-images of approximations (*i.e.* ψ such that `kind(`ψ`)` $=$ Appr) so as to ensure they will be checked before expanding too much of the original formulas. Instead of restarting the algorithm from scratch, a possible improvement that we did not implement, is to keep as much information as possible from the previous run. It is indeed possible, yet costly, to maintain dependency information at run time to retain the nodes of $\mathcal{V}$ and $\mathcal{Q}$ that are not affected by the wrong approximation. BRAB is distributed in the open source model checker Cubicle.

V. Experimental results

A. Experiments

We have evaluated our implementation of the BRAB algorithm on challenging mutual exclusion algorithms, fault-tolerance and cache coherence protocols. In the table Figure 4, we compare the performance of Cubicle when using a classical backward reachability loop (second column), and when using BRAB (first column). We run BRAB with an unlimited depth forward exploration for two processes in all benchmarks excepted for the different versions of FLASH. We also include the results for an enumerative model checker CMurphi 5.4.9 [35] and different parameterized model checkers (MCMT 2.0 [20], PFS [23], Undip [37]) to show that the examples we chose are far from trivial.

For each tool we report the execution times obtained with the best setting we found. T.O. indicates a timeout if a tool didn't answer within 24 hours and O.M. means that it exceeded a memory limit of 20 GB. For CMurphi, we give the time used to prove the safety of each benchmark for a fixed number of processes between parentheses. The last columns gives the maximum number of processes we were able to reach with 20 GB. We denote by / benchmarks that we were unable to easily translate due to syntactic restrictions. For instance PFS does not allow the update of multiple local variables at the same time, Undip does not allow variables of the type of processes and MCMT doesn't support systems with more than 50 transitions or multi-dimensional arrays.

All benchmarks were executed on a 64 bits machine with an Intel® Xeon® processor @ 3.2 GHz and 24 GB of memory. The source code for Cubicle and its implementation of the BRAB algorithm as well as all the detailed benchmarks are available at `http://cubicle.lri.fr/fmcad2013`.

In this table, Szymanski_at (resp. Szymanski_na) is an atomic (resp. non-atomic) version of Szymanski's mutual exclusion algorithm. German_Baukus is a version of Steven German's protocol extracted from [7]. German.CTC is the version translated from Ching-Tsun Chou's Murφ models, adding data paths to German_Baukus. German_pfs is an encoding of the German that was extracted from the distribution of PFS [23] where invalidation is performed non atomically. Chandra-Toueg is a version of Chandra and Toueg's reliable broadcast protocol [8] with the send omission failures model extracted from [4].

These experiments show that BRAB is very efficient on examples from the literature and is several orders of magnitude faster than backward reachability on almost all benchmarks.

B. The FLASH Cache Coherence Protocol

The Stanford FLASH (FLexible Architecture for SHared memory) multiprocessor [27] is a modular architecture designed to scale to thousands of processing units. Each processor maintains a local cache memory, whose coherence is ensured by a message passing protocol on a point-to-point network with arbitrary latency. Each memory address is *owned* by a processing unit called Home (the physical location of the given memory address).

Related work. The first proof was performed by Park and Dill [34] in 1996 using the PVS proof assistant but it requires to construct inductive invariants by hand and detail the proof steps in the assistant. This protocol was also verified by Das, Dill and Park [15] using a manually guided predicate abstraction. The method of *compositional model checking* and *data type reduction* developed by McMillan [30] which first relied on BDD based model checking in SMV was later elaborated by Chou, Mannava and Park in a framework called CMP [9]. The CMP method, formalized by Krstiċ [26], works by iteratively *abstracting* a protocol and *strengthening* the invariants from the analysis of counterexamples produced by the model checker. As of today, it is the method that scales best but it requires a lot of expert knowledge and manual intervention to devise *non interference lemmas* from counterexamples. In 2008, Talupur and Tuttle came up with the insight of using message flows conceived by protocol designers as a source of potential invariants to help the CMP method converge faster [33], [38]. Although their method is able to automatically

	BRAB	Cubicle	MCMT	PFS	Undip	CMurphi		
Szymanski_at	**0.14s**	0.30s	0.29s	T.O.	32.1s	8.04s (8)	5m12s (10)	2h50m (12)
Szymanski_na	**0.19s**	T.O.	/	/	/	0.88s (4)	8m25s (6)	7h08m (8)
German_Baukus	**0.25s**	7.03s	33m15s	/	9m43s	0.74s (4)	19m35s (8)	4h49m (10)
German.CTC	**0.29s**	3m23s	T.O.	/	/	1.83s (4)	43m46s (8)	12h35m (10)
German_pfs	**0.34s**	3m58s	5m58s	36m05s	T.O.	0.99s (4)	22m56s (8)	5h30m (10)
Chandra-Toueg	**2m17s**	2h01m	49m25s	/	/	5.68s (4)	2m58s (5)	1h36m (6)
Flash_nodata	**0.36s**	O.M.	/	/	/	4.86s (3)	3m33s (4)	2h46m (5)
Flash	**5m40s**	O.M.	/	/	/	1m27s (3)	2h15m (4)	O.M. (5)

Fig. 4: Benchmarks

generate invariant candidates from message flows, the CMP method still requires adding extra hand crafted non interference lemmas to achieve convergence on the German and FLASH protocols.

We have modeled this protocol in Cubicle's input language from the Murφ models by Ching-Tsun Chou that were used in [9], [38]. These models only account for one memory line but they (and their properties) are straightforwardly generalizable to an arbitrary number of memory addresses. The only difference we introduced is that we abstracted away the `Home` processor and for all arrays indexed by `Home`, each occurrence of `A[Home]` was replaced by a global variable `A_home`. The control property we want to verify for FLASH is

$$\forall x, y.\ x \neq y \Rightarrow \\ \texttt{CacheState}[x] = \mathsf{Exclusive} \Rightarrow \texttt{CacheState}[y] \neq \mathsf{Exclusive}$$

and the data properties are

$$\forall x.\ \texttt{CacheState}[x] = \mathsf{Exclusive} \Rightarrow \texttt{Data}[x] = \texttt{Currdata} \\ \forall x.\ \texttt{CacheState}[x] = \mathsf{Shared} \ \wedge\ \texttt{Collecting} \\ \quad \Rightarrow \texttt{Data}[x] = \texttt{PrevData} \\ \forall x.\ \texttt{CacheState}[x] = \mathsf{Shared} \ \wedge\ \neg\texttt{Collecting} \Rightarrow \\ \quad \texttt{Data}[x] = \texttt{CurrData}$$

where `CurrData`, `PrevData` and `Collecting` are auxiliary variables introduced only to specify these data properties.

We show the results obtained with Cubicle ran with option `-brab` on different versions of the FLASH protocol in Figure 5. The line nodata gives the result when we only asked to verify control properties whereas in line Flash we asked to verify both control and data properties. Finally we give the results on a version of FLASH where we introduced an error in line buggy. On this version, Cubicle exhibits an error trace highlighting a buggy behaviour.

	Forward				Backward			Total
	k	$d_{\max}$	$\lvert\mathcal{M}\rvert$	time	$\lvert\mathcal{V}\rvert$	# inv	$\lvert\mathcal{B}\rvert$	time
nodata	2	6	439	0.02s	37	30	0	**0.36s**
buggy	2	6	445	0.02s	228	/	0	**2.97s**
Flash	3	14	452,523	8.54s	1047	131	0	**5m40s**

Fig. 5: Verification of FLASH with Cubicle

Below are a few of the invariants inferred for the FLASH: they are not trivial although each one of them connects the values of only two or three variables.

$$\begin{array}{ll} (\mathrm{Inv}_1) & \neg\texttt{Invmarked}[\texttt{Home}] \\ (\mathrm{Inv}_2) & \texttt{CacheState}[\texttt{Home}] = \mathsf{Shared} \Rightarrow \\ & \quad (\texttt{Dir\_Local} \vee \neg\texttt{Dir\_Pending}) \\ (\mathrm{Inv}_3) & \forall x.\ \texttt{CacheState}[x] = \mathsf{Exclusive} \Rightarrow \texttt{Dir\_Dirty} \\ (\mathrm{Inv}_4) & \forall x.\ \texttt{CacheState}[x] = \mathsf{Exclusive} \Rightarrow \\ & \quad (x = \texttt{Home} \Rightarrow \neg\texttt{Dir\_HeadVld}) \ \wedge \\ & \quad (x \neq \texttt{Home} \Rightarrow \texttt{Dir\_HeadVld}) \end{array}$$

VI. Related Work

Parameterized verification being a largely studied problem, we focus here on the most closely related work.

Invariant generation: Ghilardi and Ranise describe in [19] an invariant synthesis algorithm for array-based systems. Their backward reachability analysis always computes precise formulas and their mechanism of guessing candidate invariants guided by the goal is similar to `BWDA` but the candidates are only filtered by syntactic heuristics. The main difference with our approach is that candidates are model checked one at a time in a completely independent resource limited backward reachability loop. Other approaches for generating inductive invariants include network invariants [21] which uses finite automata learning algorithms and split invariants [32] which connects small-model properties, inductive methods and compositional reasoning.

Cutoffs: The method of invisible invariants [6], [36] aims at discovering inductive invariants for parameterized systems that are checked up to a certain *cutoff* value obtained with a syntactic criterion. Similarly to our approach, it extracts information from a forward exploration of a finite instance of the original system. This information is generalized and, contrary to our technique, must amount to inductive invariants, whereas we only use it as an oracle. Although in Section II $I_1 \wedge I_2 \wedge I_3 \wedge \neg\Theta$ is an inductive invariant, it is generally not the case. In [17], finite instances are also used in conjunction with a template mechanism to obtain formulas that describe interesting system behaviors. Approaches based on *cutoff* and *small model* properties have been most successful when the value is detected dynamically such as in [25] although their method only works for petri-nets, and most recently in [3] which is capable of handling multiple process topologies (arrays, rings, trees, multisets) whereas our implementation of BRAB for array based transition systems only applies for linear topologies (and multisets) but scales for the Flash.

Abstraction: Abdulla *et al.* propose in [1], [2] versions of backward reachability analysis with approximated transitions. Other methods for parameterized verification are based on abstraction: the method of indexed predicates [28] automatically infers quantified predicated from which the technique of

predicate abstraction is able to construct inductive invariants: the tool UCLID which implements this technique is able to verify the German protocol [29] but not the FLASH, counter abstraction [16] whose idea is to keep track of the number of processes that satisfy a given property, and environment abstraction [11] which combines predicate abstraction with counter abstraction. In our case, we do not abstract the original system, abstractions are performed on the fly.

VII. Conclusion and Perspectives

We have presented a novel backward reachability algorithm with approximations and backtracking to check safety properties of parameterized systems. Given a correct backward reachability algorithm, we have proved the correctness of our extension. We believe that small instances of the original problem already exhibit behaviors that constitute a valuable source of knowledge. Our algorithm uses this information to filter approximations which are then model checked altogether, allowing a factoring of the proofs. It can be seen as a technique for automatically inferring invariants. We provide an open source implementation BRAB and have demonstrated the viability of our approach on several examples from the literature and FLASH, a near industrial cache coherence protocol.

An immediate line of future work is to experiment this approach on real industrial protocols such as Intel's LCP or hierarchical cache coherence protocols. While satisfactory, we think that the backtracking mechanism can be improved and that other oracles can be used for the exploration of the finite instance. Finally we would also like to explore the idea of approximations guided by finite instances in other frameworks.

Acknowledgment

This work was partially supported by the French ANR project ANR-12-INSE-0007 Cafein.

References

[1] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers. In *TACAS*. Springer, 2007.

[2] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*. Springer, 2007.

[3] P. A. Abdulla, F. Haziza, and L. Holík. All for the price of few. In *VMCAI*, pages 476–495, 2013.

[4] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Automated support for the design and validation of fault tolerant parameterized systems: a case study. *ECEASST*, 35, 2010.

[5] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, May 1986.

[6] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, pages 221–234. Springer, 2001.

[7] K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *VMCAI*, pages 317–330. Springer, 2002.

[8] T. D. Chandra and S. Toueg. Time and message efficient reliable broadcasts. In *Distributed algorithms*, pages 289–303. Springer, 1991.

[9] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*, pages 382–398. Springer, 2004.

[10] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *LICS*, pages 353–362. IEEE Press, 1989.

[11] E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *VMCAI*, pages 126–141. Springer, 2006.

[12] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *PODC'86*. ACM.

[13] A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. *Form. Methods Syst. Des.*, 34(2):104–125, Apr. 2009.

[14] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems. In *CAV*, pages 718–724. Springer, 2012.

[15] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV*, pages 160–171. Springer, 1999.

[16] E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *LICS*, pages 70–80. IEEE, 1998.

[17] M. Emmi, R. Majumdar, and R. Manevich. Parameterized verification of transactional memories. In *PLDI*, pages 134–145. ACM, 2010.

[18] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, July 1992.

[19] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *LMCS*, 6(4), 2010.

[20] S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In *IJCAR*, pages 22–29, 2010.

[21] O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *IJCAR*, pages 483–497. Springer, 2006.

[22] O. Grumberg and H. Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer-Verlag, Berlin, Heidelberg, 2008.

[23] N. B. Henda and A. Rezine. The PFS prototype model checker. http://www.it.uu.se/research/docs/fm/apv/tools/pfs/.

[24] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[25] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, pages 645–659, 2010.

[26] S. Krstić. Parametrized system verification with guard strengthening and parameter abstraction. In *AVIS*, 2005.

[27] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *ISCA*, pages 302–313. IEEE, 1994.

[28] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, pages 267–281. Springer, 2004.

[29] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.

[30] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME*, pages 179–195. Springer, 2001.

[31] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, pages 413–427. Springer, 2008.

[32] K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, pages 299–313, 2007.

[33] J. W. O'Leary, M. Talupur, and M. R. Tuttle. Protocol verification using flows: An industrial experience. In *FMCAD*. IEEE, 2009.

[34] S. Park and D. L. Dill. Protocol verification by aggregation of distributed transactions. In *CAV*, pages 300–310. Springer, 1996.

[35] G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Exploiting transition locality in automatic verification of finite-state concurrent systems. *STTT*, 6(4):320–341, 2004.

[36] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97. Springer, 2001.

[37] A. Rezine. UNDIP. http://www.it.uu.se/research/docs/fm/apv/tools/undip.

[38] M. Talupur and M. R. Tuttle. Going with the flow: Parameterized verification using message flows. In *FMCAD*, pages 1–8. IEEE, 2008.