FAR-Cubicle - A new reachability algorithm for Cubicle

Sylvain Conchon*†   Amit Goel‡   Sava Krstić§   Rupak Majumdar¶   Mattias Roux*

*LRI, Université Paris Sud CNRS, Orsay F-91405
†INRIA Saclay – Ile-de-France, Orsay cedex, F-91893
‡Apple
§Intel Corporation
¶Max Planck Institute for Software Systems

Abstract—We present a fully automatic algorithm for verifying safety properties of parameterized software systems. This algorithm is based on both IC3 and Lazy Annotation. We implemented it in Cubicle, a model checker for verifying safety properties of array-based systems. Cache-coherence protocols and mutual exclusion algorithms are known examples of such systems. Our algorithm iteratively builds an abstract reachability graph refining the set of reachable states from counter-examples. Refining is made through counter-example approximation. We show the effectiveness and limitations of this algorithm and tradeoffs that results from it.

1. Introduction

We describe FAR (Forward Abstracted Reachability), an algorithm for fully automatic verification of parameterized software systems. A parameterized system describes a family of programs such as cache coherence protocols where the number of processes involved can change but the algorithm handling their behaviour is the same for all of them. Thus, the parameter allows to talk about these algorithms without knowing the actual number of processes that will be involved and then to prove its safety regardless of this number.

Safety properties state that "nothing bad happens" in our parameterized system. Verifying them can be reduced to finding an invariant of it. Building an invariant can be hard (and even undecidable [1]). The standard approach to do so is to find a formula Φ such that Φ is an inductive invariant of the system (*i.e.* the initial state of the system satisfies Φ and taking a transition from a state satisfying Φ leads to another state satisfying Φ).

In this paper we describe an algorithm for the automatic construction of inductive invariants for array-based systems (Section 2). This algorithm, based on both IC3 [2] and Lazy Abstraction [3], builds an inductive invariant by unwinding a graph (Section 3) building a forward abstract reachability of our system. This unwinding is described as a set of non-deterministic rules. We then provide an implementation in Cubicle [4], [5], [6] (Section 5) of these rules and test its effectiveness on several cache coherence protocols (Section 6).

2. Array-based Systems

An array-based system is described in [7] as first-order logic formulas on arrays. Such a system can be described as a set of basic types, a set X of *system variables* associated to type (built as usual with basic types and standard constructions), a formula *init* representing the initial states and a set Δ of transition rules $\tau_i(X, X')$ (X' is the set X where all the variables are primed which represents the *next state* reached after the application of a transition). Since we work on *parameterized* programs, our arrays are indexed by an infinite type *proc*.

We describe the Dekker mutual exclusion algorithm as an array-based system. Each process has two boolean variables, *want* (stating that the process wants to enter in critical section or not) and *crit* (stating that the process is in critical section or not). There is a global variable *turn* of type *proc* that tracks which process can go into the critical section. Since we work in the array-based systems fragment, we represent the local variables as arrays indexed by processes and containing booleans. The set X contains two arrays, *want[proc] : bool* and *crit[proc] : bool* and the global variable *turn : proc*. Initially, no process is or wants to be in critical section. Three transitions can be triggered, one to require an access to the critical section, one to enter in it and one to exit it. According to the previous description, we write this algorithm as in Figure 1.

Since we focus on safety problems (*nothing bad happens*), we need to define what is considered as *bad states*. For Dekker algorithm, these states would be defined with the following formula :

$$\mathcal{U} \equiv \exists p_1,\ p_2.\ p_1 \neq p_2\ \wedge\ \mathtt{crit}[p_1]\ \wedge\ \mathtt{crit}[p_2]$$

Our goal is then to prove that no state represented by $\mathcal{U}$ is reachable from *init* (which can be formulated as : there exists no path $init = X_0 \xrightarrow{p_1} X_1 \xrightarrow{\cdots} \ldots \xrightarrow{p_n} X_n = \mathcal{U}$ with $p_i \in \{req, enter, exit\}$). To do so on parameterized systems, one of the main algorithms came from Ghilardi et al. with MCMT [8]. It builds the set of all reachable states by *backward reachability* (starting, then, from the unsafe state) and checks if this set contains an initial state. In this paper,

$init : \forall p.$ $\neg \texttt{want}[p] \wedge \neg \texttt{crit}[p]$

$req : \exists p.$ $\begin{aligned} &\neg \texttt{want}[p] \\ &\texttt{want}'[p] \end{aligned}$

$enter : \exists p.$ $\begin{aligned} &\texttt{want}[p] \wedge \texttt{turn} = p \\ &\texttt{crit}'[p] \end{aligned}$

$exit : \exists p_1, p_2.$ $\begin{aligned} &\texttt{crit}[p_1] \\ &\neg\texttt{want}'[p_1] \wedge \neg\texttt{crit}'[p_1] \\ &\wedge \texttt{turn}' = p_2 \end{aligned}$

Figure 1. Dekker algorithm as array-based system

we implement a different algorithm which offers a wider range of possibilities in terms of reachabilty construction.

3. Program unwinding

This algorithm also starts from $\mathcal{U}$ but tries to build an invariant of the system that does not contain it. Before going into details, we give a brief explanation. This invariant is iteratively built as an inductive invariant Θ that does not contain $\mathcal{U}$:

- if $\Theta \wedge \Delta \wedge \neg\Theta'$ is unsatisfiable then we found an inductive invariant
- if $\Theta \wedge \Delta \wedge \neg\Theta'$ is satisfiable, our candidate invariant is not inductive and we try to refine it until we either discover that there is no such refinement or we find some.

For Dekker algorithm, for example, let's take $\Theta = \neg\mathcal{U} = \forall p_1 \neq p_2. \neg crit[p_1] \vee \neg crit[p_2]$:

- $\Theta \wedge \Delta \wedge \neg\Theta'$ is satisfiable (if we take, for example, the following state : $\varphi_1 = crit[p_1] \wedge want[p_2] \wedge turn = p_2 \wedge \neg crit[p_2]$, $\varphi_1 \models \Theta$ but if we apply $enter$ to it we obtain the state $\varphi_2 = crit[p_1] \wedge crit[p_2] \wedge \ldots$ and $\varphi_2 \not\models \Theta$.)
- We need to create $\Theta'' = \Theta \wedge \rho$ which is a refinement of Θ that does not contain φ_1.

To do so, we build an *unwinding* of the algorithm as a quadruple $\langle V, E, \mathcal{W}, \mathcal{B} \rangle$, where:

- $\langle V, E \rangle$ is a rooted graph with edges labeled by transitions from Δ;
- $\mathcal{W}$ associates a formula (called *world of the vertex*) to each vertex;
- $\mathcal{B}$ associates a formula (called *bad part of the vertex*) to each vertex.

This graph contains three initial vertices :

ϵ : the root vertex, $\mathcal{W}(\epsilon) = init$ and $\mathcal{B}(\epsilon) = \perp$;
β : the unsafe vertex, $\mathcal{W}(\beta) = \top$ and $\mathcal{B}(\beta) = \mathcal{U}$;
ω : the sink vertex, $\mathcal{W}(\omega) = \perp$ and $\mathcal{B}(\omega) = \perp$.

We define $V^\epsilon = \{v \in V, \ \epsilon \xrightarrow{*} v \in E\}$ (*i.e.* the set of vertices that are linked to the root) and $(F \models_f G) \equiv (F \wedge f \models G)$

The idea behind this unwinding it that if we manage to create a graph $\mathcal{G}$ of a system $\mathcal{S} = \langle init, \Delta \rangle$ where every vertex in V^ϵ does not contain a bad part and from which no more transitions can be taken, then the disjunction of their worlds ($\Theta = \bigvee_{v \in V^\epsilon} \mathcal{W}(v)$) is an invariant of the system ($init \models \Theta$ and $\Theta \models_\Delta \Theta$).

We now propose a set of non-deterministic rules for building this unwinding. Let $\langle X, init, \Delta, \mathcal{U} \rangle$ be an array-based system. Initially, $\mathcal{G}$ is defined as follow :

- $V = \{\epsilon, \omega, \beta\}$
- $E = \emptyset$

The unwinding works by the non-deterministic application of the following rules :

Rule 1 (Extend). *If $\exists v \in V, \tau \in \Delta. \ \mathcal{W}(v) \models_\tau \top$ and $\nexists v'. v \xrightarrow{\tau} v' \in E$ then $E = E \cup \{v \xrightarrow{\tau} \beta\}$*

Rule 2 (Refine). *If $\exists v, v' \in V, \tau \in \Delta. \ v \xrightarrow{\tau} v' \in E$, $\mathcal{B}(v') \neq \perp, \exists \varphi. \mathcal{W}(v) \models_\tau \varphi$ and $\varphi \models \neg\mathcal{B}(v')$ then we create a new vertex v'' such that $\mathcal{W}(v'') = \mathcal{W}(v') \wedge \varphi$ and $E = E \cup \{v \xrightarrow{\tau} v''\} \setminus \{v \xrightarrow{\tau} v'\}$*

Rule 3 (Propagate). *If $\exists v, v' \in V, \tau \in \Delta. \ v \xrightarrow{\tau} v' \in E$, $\mathcal{B}(v') \neq \perp, \exists \gamma. \ \gamma \models \mathcal{W}(v)$, and $\gamma \models_\tau \mathcal{B}(v')$ then $B(v) \leftarrow \gamma$*

Rule 4 (Cover). *If $\exists v, v' \in V, \tau \in \Delta. \ v \xrightarrow{\tau} v' \in E, v'' \in V$ such that $\mathcal{W}(v'') \models \mathcal{W}(v')$ and $\mathcal{W}(v) \models_\tau \mathcal{W}(v'')$ then $E = E \cup \{v \xrightarrow{\tau} v''\} \setminus \{v \xrightarrow{\tau} v'\}$*

Rule 5 (Sink). *If $\exists v \in V, \tau \in \Delta. \ \mathcal{W}(v) \models_\tau \perp$ and $\nexists v'. v \xrightarrow{\tau} v' \in E$ then $E = E \cup \{v \xrightarrow{\tau} \omega\}$*

$\mathcal{S}$ is safe if and only if no rule can be applied to $\mathcal{G}$, an unwinding of it and $\mathcal{B}(\epsilon) = \perp$. Intuitively, since no more transitions can be taken and all the vertices connected to the root are not bad, $init$ will never be able to lead to $\mathcal{U}$.

4. Example

The example shown in Figure 2 describes the first four runs of the unwinding on the Dekker algorithm (we decided not to show ω since it just serves as a sink for the transitions that can not be taken from a vertex) :

(a) initially, the only rule that can be applied is **Extend** from ϵ (with $\mathcal{W}(\epsilon) \equiv init \equiv \forall p. \neg\texttt{want}[p] \wedge \neg\texttt{crit}[p]$) with req;

(b) then we can only apply **Refine** because $init \not\models_{req} \mathcal{U} \equiv \mathcal{B}(\beta)$. We create a new vertex called v_1;

(c) we can chose here to apply **Extend** from the new vertex with any transition. We chose to take the transition req;

(d) $\mathcal{W}(v_1) \not\models_{req} \mathcal{B}(\beta)$ and $\mathcal{W}(v_1) \models_{req} \mathcal{W}(v_1)$ so we can apply the cover rule;

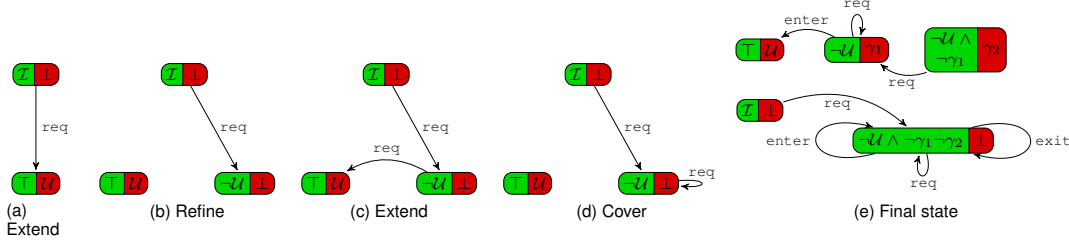(e) if we keep applying these rules, we reach a fixpoint at the third created vertex.

Figure 2. First four steps of the unwinding and final state of the graph (*sink rules are not shown*)

5. Implementation

We implemented the unwinding algorithm in Cubicle [1]. To do so we had to chose a deterministic strategy depending on multiple parameters :

- the order in which the rules are applied;
- which vertex and transition should be taken for the **Extend** rule;
- which formula φ should we take for the **Refine** rule;
- which formula γ should we take for the **Propagate** rule;
- which vertex v'' should we take for the **Cover** rule.

We came out with the following algorithm where we write $v = (W, B)$ to denote the fact that $\mathcal{W}(v) = W$ and $\mathcal{B}(v) = B$:

Algorithm 1 Graph unwinding - main loop

---

procedure FAR-CUBICLE($\mathcal{S} = \langle init, \Delta, \mathcal{U} \rangle$)
 $\epsilon \leftarrow (init, \bot)$
 $\beta \leftarrow (\top, \mathcal{U})$
 $\omega \leftarrow (\bot, \bot)$
 $V \leftarrow \{\epsilon, \beta, \omega\}$
 $E = \emptyset$
 PUSH($\mathcal{Q}, \epsilon$)   ▷ $\mathcal{Q}$ is a priority queue
 while NOT_EMPTY($\mathcal{Q}$) **do**
  $v \leftarrow$ POP($\mathcal{Q}$)
  for all $\tau \in \Delta$ **do**
   if $\mathcal{W}(v) \models_\tau \top$ **then**
    $E = E \cup \{v \xrightarrow{\tau} \beta\}$
    UNWIND($v \xrightarrow{\tau} \beta$)
   else $E = E \cup \{v \xrightarrow{\tau} \omega\}$
 return safe

This algorithm picks a vertex v from a priority queue (which initially contains only the root vertex) and for all the transitions, adds an edge to the graph from this transition to the sink vertex if the formula represented by v is inconsistent with the transition or to the unsafe vertex if the transition can be taken. If the edge goes to a vertex v' that is not the sink, the procedure UNWIND is called on it. This procedure checks if $\mathcal{B}(v) \neq \bot$ or if the $\mathcal{B}(v') = \bot$ and if both these conditions are false it tries to *close* the edge. An edge is *closed* if :

Algorithm 2 Graph unwinding - unwinding procedure

---

procedure UNWIND($v \xrightarrow{\tau} v'$)
 if $\mathcal{B}(v) = \bot \ \wedge \ \mathcal{B}(v') \neq \bot$ **then**
  switch CLOSE($v \xrightarrow{\tau} v'$) **do**   ▷ See Algorithm 3
   case Covered v"
    $E = E \cup \{v \xrightarrow{\tau} v''\} \setminus \{v \xrightarrow{\tau} v'\}$
    UNWIND($v \xrightarrow{\tau} v''$)
   case Bad φ
    if $v = \epsilon$ **then return** unsafe
    else
     $\mathcal{B}(v) \leftarrow \varphi$
     for all $u \xrightarrow{\tau'} v$ **do**
      UNWIND($u \xrightarrow{\tau} v$)
   case Refined v"
    $E = E \cup \{v \xrightarrow{\tau} v''\} \setminus \{v \xrightarrow{\tau} v'\}$
    PUSH($\mathcal{Q}, v''$)

- $\mathcal{W}(v) \models_\tau \mathcal{B}(v')$. In this case, all the edges coming to it must be unwound again;
- there exists another vertex v'' such that $v \models_\tau v''$ and $\mathcal{W}(v'') \models \mathcal{W}(v')$. In this case, the edge from v to v' is deleted and a new one from v to v'' is created and unwound;
- $\mathcal{W}(v) \not\models_\tau \mathcal{B}(v')$. A counter example φ is found a new node v'' is created with $\mathcal{W}(v'') \equiv \mathcal{W}(v') \wedge \varphi$ and pushed in the queue.

If all the edges are closed and the queue is empty, the system is safe. If the propagation of bad parts reaches the root vertex, the system is unsafe.

Algorithm 3 Graph unwinding - closing edge procedure

1: **procedure** CLOSE($v \xrightarrow{\tau} v'$)
2: **if** $\exists v''. \ \mathcal{W}(v'') \models \mathcal{W}(v') \ \wedge \ \mathcal{W}(v) \models_\tau \mathcal{W}(v'')$ **then**
3:   **return** Covered v''
4: **else if** $\mathcal{W}(v) \models_\tau \mathcal{B}(v')$ **then**
5:   **return** Bad PRE($\mathcal{B}(v'), \tau$)
6: **else**
7:   $v'' \leftarrow (\mathcal{W}(v') \ \wedge \ \text{GENERALIZE}(\neg \mathcal{B}(v')), \bot)$
8:   **return** Refined v"

As we can see on line 5 of the CLOSE procedure, the formula γ chosen for the **Propagate** rule is the pre image of the bad formula of the vertex v'. Also, on line 7 of the CLOSE procedure, the formula φ chosen for the **Refine** rule

is a generalization of the negation of the bad part of the vertex v'. These are, of course, implementation choices. Other implementation could involve *model finding*, *interpolants* ... In our case, the generalization is a naive one consisting in taking the smallest part of the resulting formula that was not already taken and that still satifies the conditions of the **Refine** rule.

6. Benchmarks

We compared our implementation to the backward reachability algorithm already implemented in Cubicle (without the invariants inference implemented with BRAB [4], [6]) and obtained the following results (the timeout was set to 5 minutes and the α version uses an abstraction engine related to the approximation implemented in BRAB to get better refinement):

Protocol	Cubicle	FAR	FAR-α
`dekker`	0.04s	0.04s	0.03s
`mux_sem`	0.04s	0.05s	0.03s
`german-ish`	0.06s	0.1s	0.55s
`german-ish2`	0.13s	0.11s	0.65s
`german-ish3`	1.2s	8.3s	0.65s
`german-ish4`	3.5s	2.5s	0.75s
`german-ish5`	1.9s	8.2s	0.60s
`german`	18s	5.8s	4.25s
`szymanski_at`	TO	13s	2.60s
`szymanski_na`	TO	TO	16s

As we can see in this table, this algorithm is competitive and even better when good refinements can be found.

7. Related Works

There has been a lot of research in software model checking and Property-Driven Reachability. This type of algorithm was first introduced by Bradley in [2] and McMillan revisited his Lazy Annotation (which shares similarities with PDR algorithms) in [9] or the recent approach from Cimatti et al. [10] and Z3 with a PDR approach in [11] and [12]. Even though some of these tools are supposed to work on parameterized systems, we were either not able to find them or they were not able to prove our examples.

8. Conclusion

We presented the problem of parameterized protocol verification and gave an algorithm to automatically do it. This new algorithm was implemented in Cubicle and successfully applied to many cache coherence protocols.

This algorithm could be improved with a better generalisation engine (allowing to explore less vertices), an incremental approach (the parameterized aspect of our language makes it hard to *remember* the state of our SMT solver). Other optimizations could involve a novel way of refining our formulas (it is clear that the best refinements are inductive invariants but it is still an open problem as how to find these).

Some optimizations were not documented in this article such as

- *Set-theoretic test* : some formulas are trivially unsatisfiable and don't require call to the SMT solver;
- *relevant instantiations* : handling universally quantified formulas can lead to multiple useless instantiations that are trivially unsatisfiable or valid and do not help the SMT solver to solve the whole formula. This optimization allows to gain a significant time in the SMT solver.
- *selecting good bads* : handling bad parts from the ones with less processes involved allows to control the number of processes that have to be instantiated when checking the satisfiability of formulas. It is mandatory, if we want to have a competitive algorithm, that we handle the bad parts cleverly (this can be done in the priority queue).

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *LICS*, 1996.

[2] A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, 2011, pp. 70–87.

[3] K. L. McMillan, "Lazy abstraction with interpolants," *CAV*, pp. 123–126, 2006.

[4] A. Mebsout, "Inférence d'invariants pour le model checking de systèmes paramétrés," Ph.D. dissertation, 2014.

[5] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi, "Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems - Tool Paper," in *CAV*, 2012, pp. 718–724.

[6] ——, "Invariants for finite instances and beyond," in *FMCAD*, 2013, pp. 61–68.

[7] S. Ghilardi and S. Ranise, "Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis," *LMCS*, vol. 6, no. 4, 2010.

[8] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli, "Towards SMT model checking of array-based systems," in *Automated Reasoning, 4th International Joint Conference, IJCAR 2008*, 2008, pp. 67–82.

[9] K. L. McMillan, "Lazy annotation revisited," in *Computer Aided Verification - 26th International Conference, CAV 2014*, 2014, pp. 243–259.

[10] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Infinite-state invariant checking with IC3 and predicate abstraction," *Formal Methods in System Design*, vol. 49, no. 3, pp. 190–218, 2016.

[11] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference*, 2012, pp. 157–171.

[12] K. Hoder, N. Bjørner, and L. M. de Moura, "μZ- an efficient engine for fixed points with constraints," in *Computer Aided Verification - 23rd International Conference, CAV 2011*, 2011, pp. 457–462.