Abduction of trap invariants in parameterized systems

Javier Esparza
Technical University of Munich
esparza@in.tum.de

Mikhail Raskin
Technical University of Munich
raskin@in.tum.de

Christoph Welzel
Technical University of Munich
welzel@in.tum.de

In a previous paper we have presented a CEGAR approach for the verification of parameterized systems with an arbitrary number of processes organized in an array or a ring [19]. The technique is based on the iterative computation of *parameterized invariants*, i.e., infinite families of invariants for the infinitely many instances of the system. Safety properties are proved by checking that every global configuration of the system satisfying all parameterized invariants also satisfies the property; we have shown that this check can be reduced to the satisfiability problem for Monadic Second Order on words, which is decidable.

A strong limitation of the approach is that processes can only have a fixed number of variables with a fixed finite range. In particular, they cannot use variables with range $[0, N-1]$, where N is the number of processes, which appear in many standard distributed algorithms. In this paper, we extend our technique to this case. While conducting the check whether a safety property is inductive assuming a computed set of invariants becomes undecidable, we show how to reduce it to checking satisfiability of a first-order formula. We report on experiments showing that automatic first-order theorem provers can still perform this check for a collection of non-trivial examples. Additionally, we can give small sets of readable invariants for these checks.

1 Introduction

Many distributed systems consist of an arbitrary number of processes executing the same algorithm. For every fixed number of processes the system has a finite state space, and can be verified using conventional model-checking techniques. However, this technique cannot prove that *all* instances of the system, one for each number of processes, are correct. Parameterized verification designs verification procedures for this task. It has developed a number of techniques, based, among others, on automata theory [4], decidable fragments of first-order logic [7, 31, 32], the theory of well-quasi-orders [1, 5, 20], or the theory of Vector Addition Systems [8, 14, 21, 23].

In recent work, we have investigated the problem of not only proving that a parameterized system satisfies a given safety property, but also providing an explanation of why the property holds in terms of *parameterized invariants* [9, 10, 19]. A parameterized invariant is an infinite family of invariants, typically one for each number of processes, that can be finitely described in an adequate language.

Assume for simplicity that an instance of the system consists of a tuple of processes $\langle P_0, \ldots, P_{N-1} \rangle$, each of them with states taken from a *finite* set of states Q. We consider variables as processes; for example, a boolean variable is a process with states true and false. A global state of the instance is a tuple $\langle q_0, \ldots, q_{N-1} \rangle \in Q^N$. Previous research has determined that instances with a small number of processes can often be proved correct using very simple inductive invariants, called *trap invariants*. A trap invariant is characterized by a tuple $\langle Q_0, \ldots, Q_{N-1} \rangle \in (2^Q)^N$, called a *trap*. The invariant states: for every reachable global state $\langle q_0, \ldots, q_{N-1} \rangle$ there exists at least one index $0 \leq i < N$ such that $q_i \in Q_i$. So, loosely speaking, the invariant says that the trap $\langle Q_0, \ldots, Q_{N-1} \rangle$ always remains populated. In the rest of this paper we identify a trap and its associated invariant if it is clear from the context what we refer

D. Bresolin and P. Ganty (Eds.): 12th International Symposium
on Games, Automata, Logics, and Formal Verification (GandALF 2021)
EPTCS 346, 2021, pp. 1–17, doi:10.4204/EPTCS.346.1

to. A set of traps proves a safety property if every global state satisfying all the associated trap invariants satisfies the property.

Observe that a trap can be seen as a word of length N over the alphabet 2^Q. A *parametric trap* is a regular language over 2^Q whose words are traps. The words of a parametric trap can have arbitrary length, and therefore induce invariants of different instances. In [19] we have developed a successful parameterized verification technique, using a CEGAR loop computing a refinement via traps. The loop maintains a set of parametric traps, initially empty. Each iteration considers an instance $\langle P_0, \ldots, P_{N-1} \rangle$ of the system, and consists of three steps:

- **Find**. Find traps $T_1, \ldots, T_k$ whose associated invariants prove the safety property for $\langle P_0, \ldots, P_{N-1} \rangle$. This is done using finite-state verification techniques.
- **Abduct**. Abduct each trap T_j (or at least some of them) into a parametric trap $\mathscr{T}_j \subseteq (2^Q)^*$. The abduction procedure follows from the very simple structure of traps, and its correctness is guaranteed by a theorem.
- **Check**. Check whether the current collection of parametric traps proves the safety property for *every* instance of the system. If not, move to the next instance that cannot be proven correct yet, and iterate.

  In [9, 19] it is shown that this step reduces to the satisfiability of a formula of monadic second-order logic on words, and is, therefore, decidable.

The main restriction of this approach is the fact that the set of states Q of a process cannot depend on the number N of processes. In particular, this forbids the use of variables with range $[0, N-1]$, where N is the number of processes. Using such variables as references to other processes allows to model *non-atomic* global checks which are essential for many distributed algorithms [26, 30]; assume, for example, processes that only have local communication capabilities (as opposed to e.g broadcast communication) but must implement some kind of mutual exclusion. Such variables require that a process has to store the values $\{0, \ldots, N-1\}$ in its state space. In this paper we address particularly the use of iteration variables for non-atomic global checks. Our contribution is as follows:

- We introduce a formal model allowing to describe global checks via looping over all processes, inspecting their local states. This happens non-atomically, i.e., after inspecting, say, process 2, and before inspecting process 3, process 2 may change its local state.
- We generalize the CEGAR loop of [19]. For this, we first show how to generalize the **Abduct** step to the new formal model. Second, we reduce the **Check** step to the satisfiability of a formula of a certain fragment of first-order logic.

The price to pay for the added generality is that the fragment is no longer decidable. However, we show that with the help of a theorem prover we can automatically find succinct human-readable correctness proofs for standard mutual exclusion algorithms.

Related work Most similar works on parameterized verification focus on systems communicating by rendez-vous and systems with atomic global checks. For example, this is the case for *regular model checking* [4], *parameterized Petri nets* [19], *component-based systems* [9, 10] and *model checking modulo theories* [24]. The latter work admits unbounded domains using decidable theories but, to the best of our knowledge, does not explicitly model non-atomic checks. This, however, is the main focus of the presented work: an explicit extension of the methodology of [19] to account for *non-atomic* global checks. To this end, we align our work more with the approaches of *view-abstraction* [2] and *symbolic*

scheme for over-approximations of parameterized systems [3]. In contrast to these, we see the benefit of our approach in providing *readable* explanations of safety proofs in form of first-order formulas. The drawback of our approach is a significant longer computation time to establish the desired safety properties. Moreover, the question whether the computed invariants induce the desired safety property is, in general, undecidable (cp. Theorem 2).

2 Formal model

First, we introduce our model of parameterized systems. Let Q and *Vars* be finite sets of states and variables respectively. Each variable $var \in Vars$ ranges over a finite set $Values_{var}$ of values. A *valuation* of *Vars* is a mapping assigning to each $var \in Vars$ an element of $Values_{var}$.

Every $N > 0$ defines an instance of the parameterized system, consisting of N *agents* with indices $0, 1, \ldots, N-1$. Every agent has a copy of Q as states, and maintains its own copy of the variables *Vars*. All agents start in the same initial state q_0, with the same initial valuation *IVal*.

Agents can execute two types of transitions. *Local* transitions are of the form

$$\langle origin, \langle var_1 := val_1, \ldots, var_k := val_k \rangle, target \rangle \tag{1}$$

where *origin* and *target* are states, $var_1, \ldots, var_k$ are distinct variables, and $val_i \in Values_{var_i}$ for all $1 \leq i \leq k$. Intuitively, the action allows an agent in state *origin* to set its own variables $var_1, \ldots, var_k$ to $val_1, \ldots, val_k$ and move to state *target*, all in one single atomic step.

Loop transitions allow agents to loop over all agents inspecting their variables. The inspecting agent or *inspector* first inspects the agent with index 0, then the agent with index 1, and so on, *in different atomic steps*. Formally, a loop transition has the form

$$\langle origin, \varphi, target_{succ}, target_{fail} \rangle, \tag{2}$$

where *origin*, $target_{succ}$, and $target_{fail}$ are states, and φ is a boolean combination of atoms of the form $var = val$ for $var \in Vars$ and $val \in Values_{var}$ and a predicate *self* which corresponds to the fact that the inspector currently inspects itself. The inspector conducts a sequence of atomic steps. At the j-th step the inspector checks if the current valuation of the variables of the agent with index j satisfies φ. If this is the case and $j = n-1$ the inspector moves to state $target_{succ}$, if this is the case but $j < n-1$ the inspector continues to execute the loop transition t and the next time the inspector is scheduled it checks the agent with index $(j+1)$. Either way, if the agent with index j does not satisfy φ the inspector moves to state $target_{fail}$.

Remark 1. *All our results can be easily extended to systems with a third kind of transitions of the form $\langle origin, \varphi_l, \varphi_r, target_{succ}, target_{fail} \rangle$, allowing an agent to check if the variables of its left and right neighbors satisfy φ_l and φ_r, respectively, and move to $target_{fail}$ or $target_{succ}$ depending on the result. We omit them for simplicity.*

Definition 1 (Parameterized System)**.** *A parameterized system over a set Vars of variables is a tuple $\mathscr{S} = \langle Q, q_0, IVal, \mathbb{T}_{lo}, \mathbb{T}_{lp} \rangle$ where $q_0 \in Q$ is an* initial state*, IVal is an* initial valuation*, and $\mathbb{T}_{lo}$, $\mathbb{T}_{lp}$ are finite sets of local and loop transitions, respectively.*

Semantics. A local configuration of an agent is a triple consisting of a *location*, a *valuation*, and a *pointer*. The location is either a state or a loop transition; intuitively, if the current location is a loop

transition t, then the agent is currently an inspector in the "middle" of executing t. The valuation contains the current values of the agent's variables. The pointer is only important when the current location is a loop transition t; in this case, the pointer is the index of the agent that the inspector is going to inspect next. With a small abuse of language, we say that *t points to* this index. A global configuration, or just configuration for short, is a sequence of local configurations, one for each agent.

It is convenient to describe global configurations, and the transitions between them, using a more verbose representation. We first introduce it informally with the help of an example, a very simplified version of Dijkstra's mutual exclusion algorithm [12] [1]. Then we give the formal definition.

Example 1 (Dijkstra's algorithm for mutual exclusion)**.** *A reduced version of Dijkstra's algorithm for mutual exclusion can be modeled with states $\{initial, loop, break, critical, done\}$ such that initial is the initial state, and one single boolean variable b with values $\{\top, \bot\}$ and initial value $\bot$. There is one loop transition*

$$t_{lp} = \langle loop, self \vee b = \bot, critical, break \rangle$$

and four local transitions:

$$\langle initial, \langle b \coloneqq \top \rangle, loop \rangle \qquad \langle break, \langle b \coloneqq \bot \rangle, initial \rangle$$
$$\langle critical, \langle \rangle, done \rangle \qquad \langle done, \langle b \coloneqq \bot \rangle, initial \rangle .$$

Figure 1 describes two steps between configurations of the instance of Dijkstra's algorithm with three agents. The agents have indices 0, 1, and 2. A local configuration of an agent is represented as a column vector with five components, labeled $loc, var, 0.t_{lp}, 1.t_{lp}, 2.t_{lp}$ in the picture. Component *loc* gives the current location of the agent; for example, in the global configuration at the top left of Figure 1, agent 0 is currently in state *break*, while agents 1 and 2 are currently executing t_{lp}. Component *var* gives the current value of variable b. The last three components give the next process to be inspected. More precisely, if agent i is executing t_{lp}, and the next agent it will inspect is the one with index j, then we write an $\uparrow$-symbol in the $i.t_{lp}$ component of vector j. Otherwise, we leave the component blank. In the formal definitions the absence of $\uparrow$ is denoted with $\square$. A (global) configuration is a sequence of three vectors, one for each agent. In the top step of the picture, agent 1 checks variable b of agent 0. Since the current value of the variable is $\top$, the agent moves to the failing state, which is *break*. In the bottom step of the picture, agent 1 checks variable b of agent 2. Since the current value is $\bot$ and $2 = N-1$, the agent moves to *critical*.

Formally, configurations are *slotted words*.

Definition 2 (Slotted words)**.** *Let $Sl = \{s_1, \ldots, s_k\}$ be a finite set of* slots*, and let $\Sigma_{s_1}, \ldots, \Sigma_{s_k}$ be finite alphabets, one for each slot. A* slotted word *over Sl is a finite word over the alphabet $\Sigma_{s_1} \times \cdots \times \Sigma_{s_k}$.*

Given a slotted word $w = a_0 \ldots a_{N-1}$ and $0 \leq j < N$, we let $a_j(s) \in \Sigma_s$ denote the component of slot s of a_j.

When it is clear from the context, we call a slotted word just a word.

Definition 3. *Let $\mathscr{S} = \langle Q, q_0, IVal, \mathbb{T}_{lo}, \mathbb{T}_{lp} \rangle$ be a parameterized system over $Vars = \{var_1, \ldots, var_k\}$, where $\mathbb{T}_{lp} = \{t_1, \ldots, t_\ell\}$, and let $N > 0$. The set of slots of $\mathscr{S}$ is*

$$Sl_{\mathscr{S}} = \{loc, var_1, \ldots, var_k, 0.t_1, \ldots, 0.t_\ell, \ldots, (N-1).t_1, \ldots, (N-1).t_\ell\}$$

[1] The version still ensures mutual exclusion, but offers no progress guarantees. Our experimental results also include an honest representation of Dijkstra's algorithm.

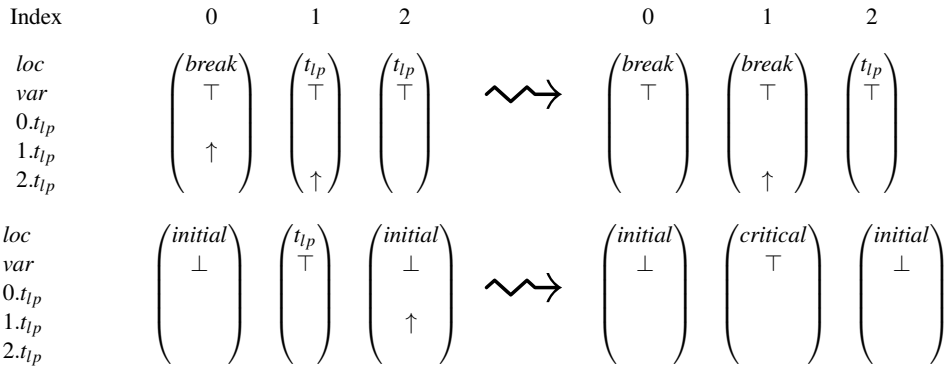

Figure 1: Steps between configurations of Dijkstra's algorithm for a 3-agent instance of Example 1.

with alphabets $\Sigma_{loc} = Q \cup \mathbb{T}_{lp}$; $\Sigma_{var_i} = Values_{var_i}$ *for every* $1 \leq i \leq k$; *and* $\Sigma_{i.t_j} = \{\uparrow, \square\}$ *for every* $0 \leq i < N$ *and* $1 \leq j \leq \ell$. *With a small abuse of language, we abbreviate the description of* $Sl_{\mathscr{S}}$ *to*

$$Sl_{\mathscr{S}} = \{loc, Vars, 0.\mathbb{T}_{lp}, \ldots, (N-1).\mathbb{T}_{lp}\}$$

A configuration *of the instance of* $\mathscr{S}$ *with N agents is a slotted word* $c_0 \ldots c_{N-1}$ *over* $Sl_{\mathscr{S}}$ *satisfying the following condition for every* $0 \leq i, j < N$ *and every* $t \in \mathbb{T}_{lp}$: $c_j(i.t) = \uparrow$ *iff* $c_i(loc) = t$ *and* $c_k(i.t) = \square$ *for every* $k \neq j$. *(Intuitively,* $c_j(i.t) = \uparrow$ *iff agent i is currently executing t and it inspects agent j next.) We call each letter* c_i *a* local configuration.

The initial configuration *of the instance is the word* c_0^N, *where* c_0 *is the local configuration given by* $c_0(loc) = q_0$, $c_0(var_i) = IVal(var_i)$ *for every* $1 \leq i \leq k$, *and* $c_0(i.t_j) = \square$ *for every* $0 \leq i < N$ *and* $1 \leq j \leq \ell$.

The semantics of the instances is given by a transition relation $\vdash$ on configurations on the basis of the transitions of $\mathscr{S}$. Its formal definition is routine, and can be found in Appendix A of the full version [17]. A configuration c is *reachable* if $c_0 \vdash^* c$, where C_0 is the initial configuration of some instance.

3 Analysis of instances

For this section, we fix a parameterized system $\mathscr{S}$ and an instance of $\mathscr{S}$ of size $N > 1$ as $\langle C_0, \vdash \rangle$ where C_0 denotes the initial configuration, and $\vdash$ its transition relation. Our analysis relies on techniques originally developed for the analysis of specific instances of a system modeled as a Petri net. The technique allows one to compute inductive disjunctive invariants – so called *traps* – of the set of reachable configurations of the instance [15]. In order to introduce them we first define *powerwords*.

Definition 4. *Let Sl be a set of slots with alphabets* Σ_s *for every* $s \in Sl$. *A* slotted powerword *is a slotted word over Sl but with alphabets* 2^{Σ_s} *for every* $s \in Sl$.

A word $w = w_0 \ldots w_{N-1}$ *and a powerword* $O = O_0 \ldots O_{m-1}$ *are* compatible *if* $m = n$. *Further, w* intersects *O, denoted* $w \sqcap O$, *if O and w are compatible and there is an index* $0 \leq i < N$ *and a slot s such that* $w_i(s) \in O_i(s)$.

Intuitively, a trap of this instance $\langle C_0, \vdash \rangle$ is a powerword of the instance satisfying the following property: the initial configuration intersects the trap, and moreover this property is inductive, i.e., the successor of a configuration intersecting the trap also intersects the trap.

Definition 5 (Trap)**.** *Let $\mathscr{S}$ be a parameterized system, and let $\langle C_0, \vdash \rangle$ be an instance of $\mathscr{S}$. A powerword O over $Sl_{\mathscr{S}}$ is a* trap *of $\langle C_0, \vdash \rangle$ if $C_0 \sqcap O$ and all configurations C, C' satisfy: if $C \vdash C'$ and $C \sqcap O$, then $C' \sqcap O$.*

Example 2. *The following powerword is a trap of the 7-agent-instance of Example 1.*

	0	1	2	3	4	5	6
loc	$\emptyset$	$\emptyset$	$\emptyset$	$\{break, loop\}$	$\emptyset$	$\{break, loop\}$	$\emptyset$
var	$\emptyset$	$\emptyset$	$\emptyset$	$\{\bot\}$	$\emptyset$	$\{\bot\}$	$\emptyset$
$0.t_{lp}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$1.t_{lp}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$2.t_{lp}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$3.t_{lp}$	$\{\uparrow\}$	$\{\uparrow\}$	$\{\uparrow\}$	$\{\uparrow\}$	$\{\uparrow\}$	$\{\uparrow\}$	$\emptyset$
$4.t_{lp}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$5.t_{lp}$	$\{\uparrow\}$	$\{\uparrow\}$	$\emptyset$	$\{\uparrow\}$	$\emptyset$	$\emptyset$	$\emptyset$
$6.t_{lp}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

First observe that the inductiveness of intersecting this powerword cannot be changed by any agent but 3 *or* 5 *executing a transition. Now, if either agent* 3 *or* 5 *executes a local transition that sets var to $\bot$ or moves into loop the resulting configuration necessarily marks this trap. If agent* 3 *(*5*) moves from state critical to state done then any configuration that intersects this trap before necessarily does not do so at the loc-slot of agent* 3 *(*5*) and, hence, the resulting configuration still intersects the trap. It remains to consider the loop transition. If agent* 3 *or* 5 *fail the loop inspection then either agent moves into state break which necessarily yields a configuration which intersects the trap. If agent* 3 *advances its pointer to some agent in $\{0,1,2,3,4,5\}$ this yields a configuration intersecting the trap. If agent* 3 *advances its pointer to agent* 6 *then agent* 5 *is inspected in this transition. Since agent* 3 *only advances if var of agent* 5 *is currently in state $\bot$ the resulting configuration intersects the trap at the var-slot of agent* 5*. If agent* 3 *inspects agent* 6 *and moves to critical only the state of agent* 3 *changes from t_{lp} to critical and the $3.t_{lp}$-slot of agent* 6*. Then, however, the resulting configuration intersects the trap at the same slot and index as the previous configuration because the occurring change is immaterial for the intersection with the trap. Analogous reasoning applies for the execution of the loop transition of agent* 5*.*

Clearly, if O is a trap of an instance $\langle C_0, \vdash \rangle$, then we have $C' \sqcap O$ for every reachable configuration C'. Therefore, a set I of traps induces an over-approximation of the set of reachable configurations, namely the set of configurations C such that $C \sqcap O$ for all $O \in I$. Algorithm 1 shows a CEGAR (*counter-example guided abstraction refinement*) loop, adapted from [15], to compute a set I of traps proving a given safety property. Starting with $I = \emptyset$, the algorithm searches for a configuration (reachable or not) that intersects all traps of I but violates the safety property. If there is none, the property holds, and the algorithm terminates; otherwise, the algorithm searches in Line 5 for a trap that is not intersected by the configuration, and adds it to I. It is well known that the set of such traps can be characterized as the solutions of a SAT formula.

4 Parameterized analysis

In the trap of Example 2, the slots $i.t_{lp}$, for $i \in \{0,1,2,4,6\}$ are *empty*: all their entries are the empty set. Intuitively, this means that the trap expresses an invariant involving only the agents 3 and 5. The important fact is that this invariant is *independent* of how many other agents there are. Indeed, Figure 2 is just one instance of a family of traps for instances of this system. Each element of this family can be represented as a slotted word with a *fixed* number of slots, independent of the number of agents. For this we first remove all the empty slots. Observe, however, that this loses information because we no longer

Algorithm 1 CEGAR loop to prove configurations unreachable.

Require: Instance $\langle C_0, \vdash \rangle$ and collection of bad configurations $\mathbb{B}$.
Ensure: All $B \in \mathbb{B}$ are unreachable from $\langle C_0, \vdash \rangle$ if the result is true.

```
if C_0 ∈ 𝔹 then
    return false
I ← ∅
while C ⊢ C' exists s.t. C ⊓ O for all O ∈ I and C' ∈ 𝔹 do
    if trap O exists with C⋪O then
        I ← I ∪ {O}
    else
        return false
return true with I
```

know which are the indices of the agents of the two nonempty rows for the loop transitions. So, we also add a new slot *index* with alphabet $\{p_0, p_1\}$ to mark these two agents. We call this process *normalization*. The result of normalizing the family at the top of Figure 2 is shown in the middle of the figure.

The normalized family can be *finitely represented* by the regular expression at the bottom of the figure. This finite expression can be seen as a *parameterized invariant* of the system. In the rest of the section we formalize the notion of a normalized trap, and present a simple *abduction procedure* that allows us to automatically generalize certain traps into regular expressions. The procedure can already be sketched in our running example. Consider, for example, the normalized slotted powerword $\mathscr{N}_0 \dots \mathscr{N}_{N-1}$ shown in the middle of Figure 2 for $2 < i$ and $i+2 < j < (N-1)-2$. Note that Example 2 shows a conceptually similar trap, although there $j = i+2$. Regardless, in Figure 2 we have $\mathscr{N}_0 = \mathscr{N}_1 = \mathscr{N}_2 = \dots = \mathscr{N}_{i-1}$ and $\mathscr{N}_{i+1} = \mathscr{N}_{i+2} = \dots = \mathscr{N}_{j-1}$. We prove that we can replace $\mathscr{N}_0\mathscr{N}_1\mathscr{N}_2$ by the regular expression $\mathscr{N}_0^+\mathscr{N}_1\mathscr{N}_2$, and similarly for $\mathscr{N}_{i+1}\mathscr{N}_{i+2}\mathscr{N}_{i+3}$, with the guarantee that all words of the regular expression are traps of the corresponding instances. Indeed, we prove that any trap with 3-repetitions of the same letter allows for this generalization. The complete process of obtaining from an actual trap a normalized one and, then, a regular expression is sketched in the three lines of Figure 2.

4.1 Normalized traps and trap languages

We now explain how to define sets of traps similar to regular languages. This requires a finite alphabet which we obtain by only paying attention to the "relevant" iteration pointers. The following definition provides the technical details.

Definition 6. *Let $\mathscr{S} = \langle Q, q_0, IVal, \mathbb{T}_{lo}, \mathbb{T}_{lp} \rangle$ be a parameterized system. Let $\mathbb{A} = \{p_0, \dots, p_{m-1}\}$ be a finite set of* agent names*. A* normalized trap *of an instance $\langle C_0, \vdash \rangle$ over the set of agent names $\mathbb{A}$ is a slotted word $\mathscr{N}_0 \dots \mathscr{N}_{N-1}$ over the set of slots*

$$\{index, loc, Vars, p_0.\mathbb{T}_{lp}, \dots, p_{m-1}.\mathbb{T}_{lp}\}$$

where the alphabet of index is $\Sigma_{index} = \mathbb{A} \cup \{\square\}$, and the other alphabets are as for normal traps, satisfying the following conditions:

- *For every name $p_i \in \mathbb{A}$ there is exactly one letter $\mathscr{N}_j$ with $\mathscr{N}_j(index) = p_i$.*
  We say that p_i is the name of the j-th agent.
- *For every name $p_i \in \mathbb{A}$ and for every loop transition $t_j \in \mathbb{T}_{lp}$ there is at least one $0 \leq k < N$ such that $p_i.t_j(k) = \{\uparrow\}$.*

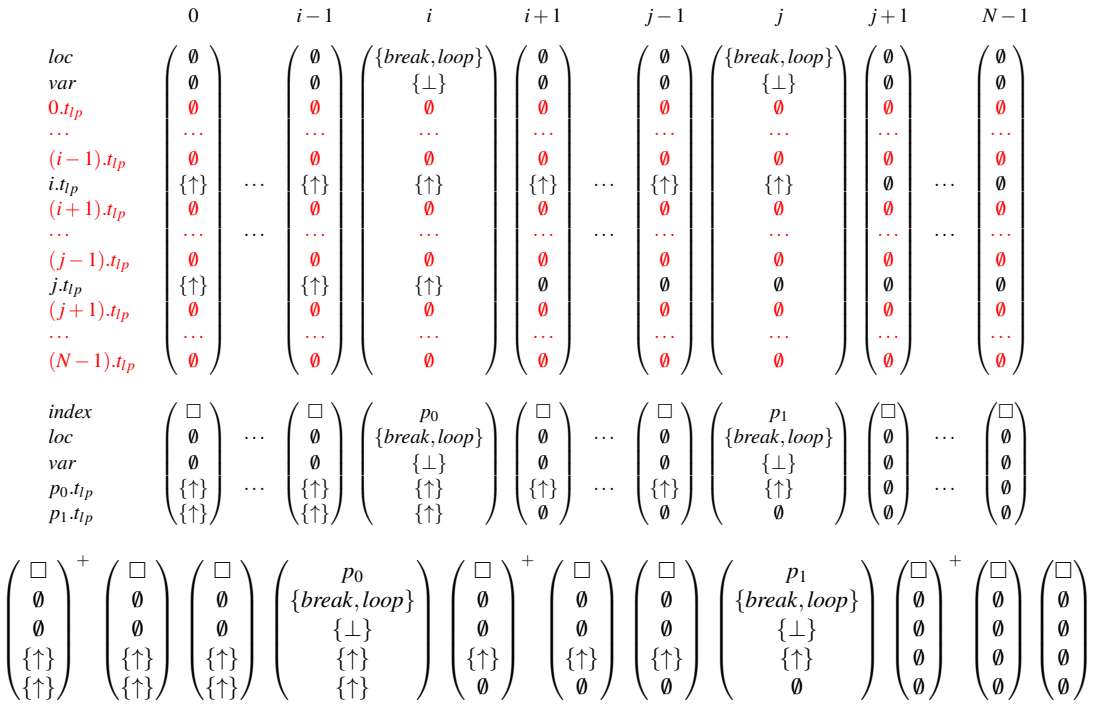

Figure 2: A family of traps, its normalization, and its finite representation as a regular expression.

- *The trap condition, as in Definition 5.*

We let $\Sigma_{\mathscr{S},\mathbb{A}}$ *denote the alphabet of normalized traps over* $\mathbb{A}$. *A* trap language *over* $\Sigma_{\mathscr{S},\mathbb{A}}$ *is a regular expression of the form* $r_0 r_1 \dots r_{\ell-1}$, *where* $r_i \in \left\{a, a^* \mid a \in \Sigma_{\mathscr{S},\mathbb{A}}\right\}$ *for every* $1 \leq i < \ell$.

The middle part of Figure 2 shows a normalized trap over $\mathbb{A} = \{p_0, p_1\}$. It is obtained by *normalizing* the trap at the top of the figure by means of the following procedure:

- Remove "empty" loop slots. Formally, if $i.t_j(k) = \emptyset$ for every $t_j \in \mathbb{T}_{lp}$ and $0 \leq k < N$, remove all the slots of $i.\mathbb{T}_{lp}$.
- Rename all remaining slots $0 \leq i_0, \dots, i_{m-1} < N$ with the "abstract names" $\{p_0, \dots, p_{m-1}\}$. Intuitively: p_k is a placeholder for some index of the word. The corresponding concrete index is marked via the additional slot *index*.
- Use the *index* slot to give the letters $\mathscr{N}_{i_0}, \dots, \mathscr{N}_{i_{m-1}}$ the names $p_0, \dots, p_{m-1}$. Label the others with $\square$.

The bottom of the figure shows the trap language obtained by normalizing all the traps shown at the top for all possible values of the indices i and j. The next theorem is the basis of the abduction process that, given a normalized trap of one instance of the system, produces, if possible, an infinite trap language of normalized traps.

Theorem 1. *Let* $\mathscr{S}$ *be a parameterized system. Let* $\mathscr{N}_0 \dots \mathscr{N}_{N-1} \in \Sigma^*_{\mathscr{S},\mathbb{A}}$ *be a normalized trap of the instance of* $\mathscr{S}$ *with N agents. If* $\mathscr{N}_i(index) = \square$ *and* $\mathscr{N}_i = \mathscr{N}_{i+1} = \mathscr{N}_{i+2}$, *then for every* $k \geq 1$ *the word*

$$\mathscr{N}_0 \dots \mathscr{N}_{i-1}\, \mathscr{N}_i^k\, \mathscr{N}_{i+1} \dots \mathscr{N}_{N-1}$$

is a normalized trap of the instance of $\mathscr{S}$ *with* $N + k$ *agents.*

Before we prove this theorem, we introduce the notions $move_k^{\mathbb{T}[\leftarrow]}$, $move_k^{\mathbb{T}[\rightarrow]}$, $drop_k$. Conceptually, $move_k^{\mathbb{T}[\leftarrow]}$ and $move_k^{\mathbb{T}[\rightarrow]}$ move pointers of the loop transitions $\mathbb{T} \subseteq \mathbb{T}_{lp}$, which are currently pointing to k, to $k-1$ and $k+1$ respectively. For example consider

$$move_1^{\{t_{lp}\}[\leftarrow]} \left(\begin{pmatrix} break \\ \top \\ \\ \uparrow \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \\ \uparrow \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \\ \end{pmatrix} \right) = \begin{pmatrix} break \\ \top \\ \\ \uparrow \\ \uparrow \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \\ \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \\ \end{pmatrix}.$$

and, analogously,

$$move_1^{\{t_{lp}\}[\rightarrow]} \left(\begin{pmatrix} break \\ \top \\ \\ \uparrow \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \\ \uparrow \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \\ \end{pmatrix} \right) = \begin{pmatrix} break \\ \top \\ \\ \uparrow \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \\ \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \\ \uparrow \end{pmatrix}.$$

Note that $move_0^{\mathbb{T}[\leftarrow]}$ is not well-defined and neither is $move_k^{\mathbb{T}[\rightarrow]}$ if k is the last index of a configuration since there is no index to move the transition pointers to.

Formally, we say $move_k^{\mathbb{T}[\leftarrow]}(c_0 \dots c_{n-1}) = c'_0 \dots c'_{n-1}$ such that $c'_\ell = c_\ell$ for all $\ell \in [n] \setminus \{k-1, k\}$. The values of variables, the current location and loop transitions that are not "moved" do not change: $c'_{k-1}(s) = c_{k-1}(s)$ and $c'_k(s) = c_k(s)$ for $s \in Vars \cup \{loc\} \cup \{i.t : t \in \mathbb{T}_{lp} \setminus \mathbb{T} \text{ and } i \in [n]\}$. Pointers in $\mathbb{T}$, however, move to the left (or to the right if we consider $move_k^{\mathbb{T}[\rightarrow]}$ instead): $c'_k(\ell'.t) = \square$ while $c'_{k-1}(\ell'.t) = \uparrow$ if either $c_{k-1}(\ell'.t) = \uparrow$ or $c_k(\ell'.t) = \uparrow$ and otherwise $c'_{k-1}(\ell'.t) = \square$ for all $\ell' \in [n]$ and $t \in \mathbb{T}$.

Secondly, $drop_k$ describes how to remove a specific index k from a configuration. Essentially, we remove all slots $k.\mathbb{T}_{lp}$ and the column k. As an example, consider

$$drop_3 \left(\begin{pmatrix} break \\ \top \\ \\ \uparrow \\ \square \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \\ \uparrow \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \square \\ \square \\ \square \end{pmatrix} \right) = \begin{pmatrix} break \\ \top \\ \\ \uparrow \end{pmatrix} \begin{pmatrix} t_{lp} \\ \top \\ \end{pmatrix}.$$

Hence, $drop_k(c_0 \dots c_{n-1}) = c'_0 \dots c'_{n-2}$ such that

- for $\ell_1 < k$, $\ell_2 < k$, $t \in \mathbb{T}_{lp}$ we have $c'_{\ell_1}(\ell_2.t) = c_{\ell_1}(\ell_2.t)$ and $c'_{\ell_1}(s) = c_{\ell_1}(s)$ for all $s \in Vars \cup \{loc\}$,
- for $\ell_1 < k$, $k \leq \ell_2$, $t \in \mathbb{T}_{lp}$ we have $c'_{\ell_1}(\ell_2.t) = c_{\ell_1}(\ell_2 + 1.t)$,
- for $k \leq \ell_1$, $\ell_2 < k$, $t \in \mathbb{T}_{lp}$ we have $c'_{\ell_1}(\ell_2.t) = c_{\ell_1+1}(\ell_2.t)$ and $c'_{\ell_1}(s) = c_{\ell_1+1}(s)$ for all $s \in Vars \cup \{loc\}$, and
- for $k \leq \ell_1$, $k \leq \ell_2$, $t \in \mathbb{T}_{lp}$ we have $c'_{\ell_1}(\ell_2.t) = c_{\ell_1+1}(\ell_2 + 1.t)$.

We make a few observations about $drop_k$:

- $drop_k$ yields a configuration if no slot of any loop transition of agent k contains the value $\uparrow$.
- Let C be a configuration and O a compatible powerword. If $c_k(s) \notin O_k(s)$ for all slots s and $c_\ell(k.t) \notin O_\ell(k.t)$ for all ℓ and $t \in \mathbb{T}_{lp}$ then $drop_k(C) \sqcap drop_k(O)$ iff $C \sqcap O$.

Equipped with these notions we prove Theorem 1:

Proof. For the sake of contradiction assume that the statement of Theorem 1 is incorrect. Then, we can fix a minimal k_0 such that the normalized word $\mathscr{N}_0 \dots \mathscr{N}_{i-1} \mathscr{N}_i^{k_0} \mathscr{N}_{i+1} \mathscr{N}_{N-1}$ gives a powerword O which is *not* a trap in the instance of $N + k_0 - 1$ agents.

Therefore, let $C = c_0 \dots c_{N-k_0-2}$ and $C' = c'_0 \dots c'_{N-k_0-2}$ be configurations of the instance with $N + k - 1$ agents such that $C \vdash C'$ and $C \sqcap O$, but $C' \not\sqcap O$.

Now, observe that $k_0 > 1$ since $k_0 = 1$ immediately contradicts with the prerequisites of the theorem. However, since k_0 is minimal, $\mathscr{N}' = \mathscr{N}_0 \dots \mathscr{N}_{i-1} \mathscr{N}_i^{k_0-1} \mathscr{N}_{i+1} \mathscr{N}_{N-1}$ *is* a normalized trap of $N + k_0 - 2$ agents. Let O' be the corresponding powerword – which is, in fact, a trap. Note that $O' = drop_i(O) = drop_{i+1}(O) = \dots = drop_{i+k_0}(O)$.

Consider the case where $C \vdash C'$ is an instance of a local transition t. Thus, C and C' differ at exactly one index j and there at most in slots from $Vars \cup \{loc\}$. Pick now $m \in \{i, i+1, i+2, i+3\} \setminus \{j\}$. W.l.o.g. we assume that $m - 1 \in \{i, i+1, i+2, i+3\}$ (otherwise exchange $m-1$ with $m+1$ and $move_m^{\mathbb{T}_{lp}[\leftarrow]}$ with $move_m^{\mathbb{T}_{lp}[\rightarrow]}$ within the following argument). By the definition of instances of local transitions it is straightforward to see that $move_m^{\mathbb{T}_{lp}[\leftarrow]}(C) \vdash move_m^{\mathbb{T}_{lp}[\leftarrow]}(C)$. However, we still have $move_m^{\mathbb{T}_{lp}[\leftarrow]}(C) \sqcap O$ but $move_m^{\mathbb{T}_{lp}[\leftarrow]}(C') \not\sqcap O$ since $O_{m-1} = O_m$ and, thus, $O_{m-1}(\ell.t) = O_m(\ell.t)$ for all $\ell \in [n + k_0 - 1]$ and $t \in \mathbb{T}_{lp}$. Since agent m does not contain any $\uparrow$ values anymore $(drop_m \circ move_m^{\mathbb{T}_{lp}[\leftarrow]})(C)$ and $(drop_m \circ move_m^{\mathbb{T}_{lp}[\leftarrow]})(C')$ indeed are configurations.

Moreover, by instantiating the same local transition for agent j or, if $m \leq j$, $j-1$ gives $(drop_m \circ move_m^{\mathbb{T}_{lp}[\leftarrow]})(C) \vdash (drop_m \circ move_m^{\mathbb{T}_{lp}[\leftarrow]})(C')$. Note that $O_\ell(m.t) = \emptyset$ for all $t \in \mathbb{T}_{lp}$ since $\mathscr{N}_m(index) = \square$ and $c_m(s) \notin O_m(s)$ for all slots s since $c_m(s) = c'_m(s)$ because $j \neq m$ and $C' \not\sqcap O$. Thus, by our observations about $drop_m$, we have $(drop_m \circ move_m^{\mathbb{T}_{lp}[\leftarrow]})(C) \sqcap O'$ because $O' = drop_m(O)$, but $(drop_m \circ move_m^{\mathbb{T}_{lp}[\leftarrow]})(C') \not\sqcap O'$ contradicting that O' is a trap.

Consider the case that $C \vdash C'$ is an instance of a loop transition t. Then there are indices j and p such that j is the agent executing the loop transition while p is the agent that j currently inspects. If the loop transition fails, p changes from C to C' only in slot $j.t$ from $\uparrow$ to $\square$ while j changes from C to C' only in slot loc from t to $target_{fail}$. In this case pick $m \in \{i, i+1, i+2, i+3\} \setminus \{j, p\}$. Again, we assume $m-1$ in $\{i, i+1, i+2, i+3\}$. Using $O_i = O_{i+1} = O_{i+2} = O_{i+3}$ and, m not being involved in this instance of t allows us to consider $D = move_m^{\mathbb{T}_{lp}[\leftarrow]}(C) \vdash move_m^{\mathbb{T}_{lp}[\leftarrow]}(C') = D'$ instead. Exploiting the same principles above – namely, the fact that the $m.t$ slot is empty for every letter in O and the agent m not changing from D to D' – allows us to deduce that $drop_m(D) \sqcap O'$ while $drop_m(D') \not\sqcap O'$ which contradicts the assumption of k_0 being minimal.

It remains to consider the case that this transition is an instance of t for inspector j and inspectee p which is successful. If p is the last agent, then C and C' only differ at indices j and p. Hence, the pattern of the before mentioned cases applies again.

Otherwise, only agents j, p, and $p+1$ change. Then, however, there is $m \in \{i, i+1, i+2, i+3\} \setminus \{j, p, p+1\}$. Repeating the same arguments as above we can show that O' cannot be a trap since $(drop_m \circ move_m^{\mathbb{T}_{lp}[\leftarrow]})(C) \vdash (drop_m \circ move_m^{\mathbb{T}_{lp}[\leftarrow]})(C')$ is an instance of a transition for the instance of size $N + k_0 - 2$. □

In the next section we show how to use a first-order theorem prover to check whether every configuration of every instance satisfying the invariants of a family of trap languages also satisfies a given safety property.

4.2 Proving safety properties

Recall that in [19] we consider parameterized systems without loop transitions, i.e., having only local transitions and the transitions described in Remark 1. For these systems, the problem whether every global configuration satisfying a family of trap languages also satisfies a desired safety property can be reduced to the satisfiability problem for *WS1S*, which is decidable (see also [9, 10]). Unfortunately, this is no longer the case for systems with loop transitions.

Let $\mathscr{S} = \langle Q, q_0, IVal, \mathbb{T}_{lo}, \mathbb{T}_{lp} \rangle$ be a parameterized system. For every $q \in Q$, let $\mathscr{C}_q$ denote the set of global configurations of all instances of $\mathscr{S}$ such that at least one agent is in state q. Similarly, for every variable x and every value v of $Values_x$, let $\mathscr{C}_{v,x}$ be the set of global configurations of all instances of $\mathscr{S}$ such that for at least one agent the variable x has value v. A *safety property* $\mathscr{P}(C)$ is a Boolean combination of the predicates $C \in \mathscr{C}_q$ and $C \in \mathscr{C}_{v,x}$. An example safety property could be "there is no agent in the state *critical*, or there is an agent in state *loop* and no agent such that $b = \top$". Let $\mathscr{L} = \{\mathscr{L}_1, \ldots, \mathscr{L}_k\}$ be a finite set of trap languages. Abusing language, let $\mathscr{L}(C)$ denote the predicate that holds for a global configuration C if $C \sqcap O$ for every trap $O \in \mathscr{L}$ compatible with C.

Given $\mathscr{S}, \mathscr{P}, \mathscr{L}$, we want to decide if every configuration C satisfying $\mathscr{L}(C)$ also satisfies $\mathscr{P}(C)$, i.e., whether the invariants of $\mathscr{L}$ are sufficient to prove the safety property $\mathscr{P}$. Since $\mathscr{L}(C)$ is an inductive predicate (i.e., $\mathscr{L}(C)$ and $C \vdash C'$ imply $\mathscr{L}(C')$), this is the case if (1) $\mathscr{P}$ holds for every initial configuration of every instance of $\mathscr{S}$, and (2) $\mathscr{L}(C) \wedge \mathscr{P}(C) \wedge C \vdash C'$ implies $\mathscr{P}(C')$. This leads to the following definition:

Definition 7. *Let $\mathscr{S}, \mathscr{L}, \mathscr{P}$ be a parameterized system, a trap language, and a safety property, respectively. The* inductivity problem *consists of deciding if $(\mathscr{L}(C) \wedge \mathscr{P}(C) \wedge C \vdash C') \rightarrow \mathscr{P}(C')$ holds for every two configurations C, C' of $\mathscr{S}$.*

Theorem 2. *The inductivity problem is undecidable.*

Sketch. Any loop transition t implicitly provides a reference from agent to agent. We can use both this and the order of agents to enforce grid structures within instances. This allows us to give a reduction of the problem whether a periodic tiling for a given set of Wang tiles exists to the question above. Then, the result follows from the undecidability of finding periodic tilings for Wang tiles [27]. For the formal details refer to Appendix B of the full version [17]. □

Due to this result, there is no hope of reducing the safety problem for systems with loop transitions to the satisfiability problem of a decidable logic. It is still possible, though, to reduce it to the satisfiability problem of first-order logic with monadic predicates and one function symbol, and run it through an automatic first-order theorem prover. Informally, we must embed the statement

$$\forall C, C'. (\mathscr{L}(C) \wedge \mathscr{P}(C) \wedge C \vdash C') \rightarrow \mathscr{P}(C')$$

in first-order logic.

Embedding into first-order logic. In [19] we consider configurations that are solely defined by the values of the local variables and the state of a process. This can be encoded by a set of monadic second-order variables; namely, we introduce a monadic second-order variable $\boldsymbol{X}_{val}^{var}$ for all values *val* of all $var \in Vars$.

In the presence of loop transitions, the local configuration also includes the pointer, whose value ranges from 0 to $N-1$, where N is the number of processes. For this we introduce one function symbol f.

This function symbol is interpreted by maps from $\{0,\dots,N-1\}$ to $\{0,\dots,N-1\}$. Combining this, we describe a configuration C as monadic predicates $\boldsymbol{X}^{var}_{val}$ for every $val \in Values_{var}$ for all $var \in Vars$, $\boldsymbol{X}_q$ for every $q \in Q$, $\boldsymbol{X}_t$ for every $t \in \mathbb{T}_{lp}$ and a unary function symbol f. It is obvious, however, that not every interpretation of these monadic predicates and the function symbol f properly describes a configuration. Only models which ensure that every variable is currently in exactly one value for every agent, and that every agent is currently in exactly one location can be considered proper. For this, we introduce the formula η which ensures these properties. Namely, we define

$$\eta = \forall i : \begin{pmatrix} \bigwedge_{var\in Vars} \bigvee_{val\in Values} \boldsymbol{X}^{var}_{val}(i) \wedge \bigwedge_{val'\in Values\setminus\{val\}} \neg \boldsymbol{X}^{var}_{val'}(i) \\ \wedge \bigvee_{val\in Q\cup\mathbb{T}_{lp}} \boldsymbol{X}_{val}(i) \wedge \bigwedge_{val'\in Q\cup\mathbb{T}_{lp}\setminus\{val\}} \neg \boldsymbol{X}_{val'}(i) \end{pmatrix}.$$

η enforces that every agent currently is in exactly one location and sets every variable to exactly one value. We introduce notions which leverage this observation to increase readability of formulas. Namely, we write $var(i) = val$ and $loc(i) = q$ ($loc(i) = t$) to express $\boldsymbol{X}^{var}_{val}(i)$ and $\boldsymbol{X}_q(i)$ ($\boldsymbol{X}_t(i)$) respectively. Moreover, we introduce $var(i) = var(j)$ as a short form of $\bigwedge_{val\in Values_{var}} \boldsymbol{X}^{var}_{val}(i) \leftrightarrow \boldsymbol{X}^{var}_{val}(j)$ and, as its dual, $var(i) \neq var(j)$ expresses $\neg(var(i) = var(j))$.

To encode the inductivity problem it is necessary to relate two configurations C and C' such that $C \vdash C'$. For this, we introduce a primed version of all monadic predicates and the function symbol f which we use to encode a configuration to represent C'. Additionally, we axiomatize a total linear order $\leq$ between the constant symbols 0 and $(N-1)$ in a formula ψ. Since ψ's definition is fairly standard we omit it here for brevity. Without increasing the expressiveness of the theory of this total linear order, we introduce $t < t'$ to express $t \neq t' \wedge t \leq t'$ and $t+1$ as a term to refer to the immediate successor of t w.r.t. $\leq$ (this requires that t cannot be interpreted with $N-1$ which is ensured for all occurrences). As ψ, the encoding of $C \vdash C'$ is now fairly standard. We postulate the existence of a formula τ which relates the predicates and the function symbols f and f' of C and C'. One clause in τ, for example, which encodes the loop transition t_{lp} of Example 1 (cp. Figure 1) is

$$\begin{aligned} &\forall \ell : (var(\ell) = var'(\ell)) \\ \wedge &\exists i : loc(i) = t \\ &\wedge \forall k \neq i : (loc(k) = loc'(k) \wedge f(k) = f'(k)) \\ &\wedge \begin{pmatrix} (f(i) \neq i \wedge var(f(i)) = \top) \rightarrow loc'(i) = break \\ \wedge((f(i) = i \vee var(f(i)) = \bot) \wedge i < (N-1)) \rightarrow loc'(i) = t \wedge f'(i) = f(i)+1 \\ \wedge((f(i) = i \vee var(f(i)) = \bot) \wedge i = (N-1)) \rightarrow loc'(i) = critical \end{pmatrix}. \end{aligned}$$

Secondly, the inductivity problem relies on only considering configurations C which intersect *all* compatible normalized traps of the computed trap languages. The trap language at the bottom of Figure 2 describes one of these trap languages. Careful analysis of Example 1, however, reveals a stronger invariant[2]. Namely, that all words in

$$\begin{array}{ll} index & \begin{pmatrix}\square\\ \emptyset\\ \emptyset\\ \{\uparrow\}\\ \{\uparrow\}\end{pmatrix}^* \begin{pmatrix} p_0\\ \{break, loop\}\\ \{\bot\}\\ \{\uparrow\}\\ \{\uparrow\}\end{pmatrix} \begin{pmatrix}\square\\ \emptyset\\ \emptyset\\ \{\uparrow\}\\ \emptyset\end{pmatrix}^* \begin{pmatrix} p_1\\ \{break, loop\}\\ \{\bot\}\\ \{\uparrow\}\\ \emptyset\end{pmatrix} \begin{pmatrix}\square\\ \emptyset\\ \emptyset\\ \emptyset\\ \emptyset\end{pmatrix}^* \\ loc & \\ var & \\ p_0.t_{lp} & \\ p_1.t_{lp} & \end{array} \tag{3}$$

[2]In fact, this is one of the invariant languages that our tool `heron` computes for this example.

are normalized traps in instances of the corresponding size. On this basis, we can derive a formula $\Phi = \eta \wedge \psi \wedge \varphi$ whose models are the global configurations C that intersect *all* traps of this language, and are compatible with C. We already discussed the nature of η and ψ. It remains to introduce φ which restricts models of Φ to configurations which intersect every compatible trap of the language. The traps of the language are characterized by the positions of the abstract names p_0 and p_1. So φ is of the form "for every two indices $0 \leq p_0 < p_1 < N$, the (unique) trap of the language corresponding to these values of p_0 and p_1 intersects C". The formula splits into several cases corresponding to having an intersection of C and the considered trap at some index $i < p_0$, at p_0, between p_0 and p_1, or at p_1. Consider the case where an intersection occurs at some index $i < p_0$: since the letters to the left of the letter with index p_0 are of the form $(\square, \emptyset, \emptyset, \{\uparrow\}, \{\uparrow\})$, this is the case if one of the agents p_0 or p_1 is currently executing its loop transition, and its pointer points to i. This is captured by the formula $t(p_0) = i$ or $t(p_1) = i$. Proceeding like this for the other cases gives

$$\varphi = \forall p_0 < p_1 \; : \; \begin{pmatrix} \exists i < p_0 \,.\, t(p_0) = i \vee t(p_1) = i \\ \vee state(p_0) = break \vee state(p_0) = loop \vee var(p_0) = \bot \vee t(p_0) = p_0 \vee t(p_1) = p_0 \\ \vee \exists p_0 < i < p_1 \,.\, t(p_0) = i \\ \vee state(p_1) = break \vee state(p_1) = loop \vee var(p_1) = \bot \vee t(p_0) = p_1 \end{pmatrix}.$$

Recall from Definition 6 that trap languages are defined as regular expressions of the form $r_0 r_1 \dots r_{\ell-1}$ where every r_i is either a single letter; i.e., some a, or the arbitrary repetition of a single letter; i.e., the expression a^*. Hence, it is straightforward to define φ for a finite collection of trap languages by generalizing the example above. Again, we avoid giving this definition for brevity.

The inductiveness problem corresponds now to the question whether

$$(\Phi \wedge \tau \wedge \mathscr{P}) \rightarrow \mathscr{P}'$$

is valid where $\mathscr{P}$ and $\mathscr{P}'$ are formalizations of the desired safety property; e.g., $\neg(\exists i \neq j : loc(i) = critical \wedge loc(j) \neq critical)$ and $\neg(\exists i \neq j : loc'(i) = critical \wedge loc'(j) \neq critical)$ for Example 1.

5 Experimental results

We implemented the approach we describe in this paper in our tool `heron` [18, 35] which is written in the Python programming language. We implemented Algorithm 1 using `clingo` [22] and relied for first-order theorem proving on VAMPIRE [29] and CVC4 [6]. Essentially, `heron` runs Algorithm 1 and normalizes the occurring traps. In these normalized traps, it identifies repetitions which can be safely generalized via Theorem 1. Additionally, we try to obtain succinct invariants. That is, languages with small regular expressions. As an example, we consider the language from Equation (3) as succinct. To this end, once `heron` encounters a repetition of letters in a normalized traps it proceeds to shrink and expand this repetition and checks whether the resulting normalized traps correspond to actual traps. If a repetition can be expanded to the generalization threshold it can be safely repeated arbitrarily often. This heuristic is relatively cheap since it simply relies on checking whether a normalized trap is indeed a trap but allows us to easily compute succinct but general trap languages.

Examples As a benchmark we used classical examples of mutual exclusion algorithms. Namely, the mutual exclusion algorithm of Dijkstra [12] (here we prove the reduced version as described in Example 1

Figure 3: The experimental results for our tool `heron`.

Algorithm	time (s)	max. N	# traps	# trap languages	max. # indices	max. proving time (s)
Example 1	11	8	36	2	2	1
Dijkstra's	22	8	75	5	2	1
Knuth's	194	8	160	7	2	1
de Bruijn's	76	8	164	6	2	1
Eisenberg & McGuire's	1055	9	126	6	2	1

as well as a precise formulation), Knuth [28], de Bruijn [11], and Eisenberg & McGuire [13]. For all these examples we can prove fully automatically that they ensure the mutual exclusion property for their respective critical sections. To the best of our knowledge these are the first fully automatic proofs for the algorithms of Knuth, de Bruijn, and Eisenberg & McGuire for a model with non-atomic global checks.

However, all but Example 1 require the use of global pointers to processes; i.e., we introduce a variable with some value in $[N]$ for every instance of size N and processes can check the current value of local variables of the process this pointer currently refers to, set their own identifier as the current value of the pointer, or upon encountering a certain state in an agent during an iteration setting the pointer to this agent. We adapt our approach to account for this: for every pointer we introduce an additional slot with a corresponding alphabet of $\{\uparrow, \square\}$ and maintain the invariant that every configuration contains exactly one agent for which a pointer slot holds the value $\uparrow$. This makes it necessary to broaden the scope of Theorem 1. Essentially, we obtain now that every transition might interact with all agents a pointer currently points to. Careful examination of the proof of Theorem 1 suggests that we mainly rely on the fact that for every transition only a finite amount of agents are essential. Adding pointer variables might increase the amount of agents that are considered by a transition. This is immaterial for the spirit of Theorem 1 though, but increases the amount of repetitions necessary to allow for safe generalization. Fortunately, as long as the necessary invariants happen to be simple, the direct performance impact is limited to checking a few more easily obtained traps for finite instances. In Appendix C of the full version [17] we shortly discuss the formalization of this observation.

Results The details of our experiments can be found in Figure 3. We report in the first column the name of the considered algorithm. The second column contains the time it took to prove the mutual exclusion property for this system. In the third column we give the maximal size for which we instantiated and analyzed the system. The fourth column reports the amount of traps that were computed for the various instances of the system while the fifth column gives the amount of trap languages that were actually necessary to establish the mutual exclusion property. `heron` axiomatizes for the inductivity problem a minimal size of the considered configurations which exceeds the already analyzed instances. This axiom, however, is not used for the proof of all examples; i.e., all inductivity queries can be solved without this axiom in comparable time. This means that the reported trap languages indeed are sufficient to establish the mutual exclusion property *for all instances*. In the sixth column we report the maximum amount of abstract indices that occur in any of the trap languages. Finally, the last column contains the maximum time it took to solve the final first-order problems for the various transitions. We observe that the proof search does not dominate the running time, the real bottleneck being the analysis of the finite instances.

The information of Figure 3 are extracted from log files that `heron` produces on its execution. We want to stress specifically that `heron` does not only automatically prove the desired property but also

gives a detailed and human-readable explanation. This explanation presents itself in form of the computed traps for considered finite instances as well as the computed trap languages which suffice to prove the inductivity problem for all instances which are larger then the analyzed finite instances. These trap languages can be read and understood by humans, and are for all considered examples sufficient to prove inductivity of the mutual exclusion property for all instances of the parameterized system. As a complete example, consider the language in Equation (3) and

$$\begin{array}{l} \mathit{index} \\ \mathit{loc} \\ \mathit{var} \end{array} \quad \begin{pmatrix} \square \\ \emptyset \\ \emptyset \end{pmatrix}^* \quad \begin{pmatrix} \square \\ \{\mathit{initial}\} \\ \{\top\} \end{pmatrix} \quad \begin{pmatrix} \square \\ \emptyset \\ \emptyset \end{pmatrix}^* \tag{4}$$

which are the only languages `heron` computes to prove Example 1. Additionally, `heron` gives a series of first-order problems in the widely adopted `TPTP` format [34] which describe the inductiveness checks of the proven property. This allows a user to leverage the advanced tool support for first-order theorem proving [6, 25, 33] to verify the proof and, simultaneously, better understand the analyzed system.

In fact, the data of Figure 3 shows that mutual exclusiveness in the critical section can be established with very few invariants (e.g., in less than 7 trap languages). Moreover, all these languages of normalized traps need at most two references to indices. Hence, the invariants express – in some sense – *local* properties. Finally, we want to draw attention to the fact that first-order theorem proving is extremely efficient once all necessary invariants are established. This suggests that the presented approach is practically viable despite Theorem 2.

6 Conclusion

In our previous work, we showed the value of structural invariants of parameterized systems for their analysis [9]. This contribution is the natural expansion of our abduction principles from [19] to parameterized systems with non-atomic global checks. Although one sacrifices the decidability of the inductivity problem in this expansion, the experimental data suggests that it remains practically viable. This contrasts observations from [9, 19] where the inductivity problem is decidable albeit it often fails in practice due to its computational complexity. Moreover, most of the time is spent on the analysis of the finite instances, while generalisations of suitable finite invariants can be verified very quickly. We therefore conclude that this expansion is worthy of consideration and, in a broader view, establishes analysis of parameterized systems via abduction of structural invariants as a promising direction. Moreover, we believe that our work complements existing approaches like *view abstraction* [2]. Especially, since the invariants `heron` computes for Dijkstra's algorithm can be similarly expressed in the formalism of view abstraction. However, `heron` obtains these languages from structural analysis of instances while in view abstraction one abstracts from the reachable set of configurations for fixed sizes. Also, `heron` computes comparatively very few invariants but needs *significantly more* time to do so. Thus, future work can focus on improving the analysis of finite instances via structural properties. Moreover, it is tempting to further explore the trade-offs between how strong an analysis is and how easy the considered structural properties can be abducted to the general case. In particular, a richer set of considered invariants should allow more diverse communication structures between the agents. Additionally, one can consider applying abduction principles to proofs of liveness properties in, e.g., Petri nets [16].

References

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson & Yih-Kuen Tsay (1996): *General Decidability Theorems for Infinite-State Systems*. In: *LICS*, IEEE Computer Society, pp. 313–321, doi:10.1109/LICS.1996.561359.

[2] Parosh Aziz Abdulla, Frédéric Haziza & Lukás Holík (2016): *Parameterized verification through view abstraction*. *Int. J. Softw. Tools Technol. Transf.* 18(5), pp. 495–516, doi:10.1007/s10009-015-0406-x.

[3] Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno & Ahmed Rezine (2008): *Handling Parameterized Systems with Non-atomic Global Conditions*. In: *VMCAI*, *Lecture Notes in Computer Science* 4905, Springer, pp. 22–36, doi:10.1007/978-3-540-78163-9_7.

[4] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson & Mayank Saksena (2004): *A Survey of Regular Model Checking*. In: *CONCUR*, *LNCS* 3170, Springer, pp. 35–48, doi:10.1007/978-3-540-28644-8_3.

[5] Parosh Aziz Abdulla, A. Prasad Sistla & Muralidhar Talupur (2018): *Model Checking Parameterized Systems*. In: *Handbook of Model Checking*, Springer, pp. 685–725, doi:10.1007/978-3-319-10575-8_21.

[6] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4*. In: *CAV*, *Lecture Notes in Computer Science* 6806, Springer, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[7] Kai Baukus, Saddek Bensalem, Yassine Lakhnech & Karsten Stahl (2000): *Abstracting WS1S Systems to Verify Parameterized Networks*. In: *TACAS*, *LNCS* 1785, Springer, pp. 188–203, doi:10.1007/3-540-46419-0_14.

[8] Michael Blondin, Javier Esparza, Martin Helfrich, Antonín Kucera & Philipp J. Meyer (2020): *Checking Qualitative Liveness Properties of Replicated Systems with Stochastic Scheduling*. In: *CAV (2)*, *LNCS* 12225, Springer, pp. 372–397, doi:10.1007/978-3-030-53291-8_20.

[9] Marius Bozga, Javier Esparza, Radu Iosif, Joseph Sifakis & Christoph Welzel (2020): *Structural Invariants for the Verification of Systems with Parameterized Architectures*. In: *TACAS (1)*, *Lecture Notes in Computer Science* 12078, Springer, pp. 228–246, doi:10.1007/978-3-030-45190-5_13.

[10] Marius Bozga, Radu Iosif & Joseph Sifakis (2021): *Checking deadlock-freedom of parametric component-based systems*. *J. Log. Algebraic Methods Program.* 119, p. 100621, doi:10.1016/j.jlamp.2020.100621.

[11] N. G. de Bruijn (1967): *Additional comments on a problem in concurrent programming control*. *Commun. ACM* 10(3), pp. 137–138, doi:10.1145/363162.363167.

[12] Edsger W. Dijkstra (2002): *Cooperating Sequential Processes*, pp. 65–138. Springer New York, New York, NY, doi:10.1007/978-1-4757-3472-0_2.

[13] Murray A. Eisenberg & Michael R. McGuire (1972): *Further Comments on Dijkstra's Concurrent Programming Control Problem*. *Commun. ACM* 15(11), p. 999, doi:10.1145/355606.361895.

[14] Javier Esparza, Pierre Ganty, Jérôme Leroux & Rupak Majumdar (2017): *Verification of population protocols*. *Acta Informatica* 54(2), pp. 191–215, doi:10.1007/s00236-016-0272-3.

[15] Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp J. Meyer & Filip Niksic (2014): *An SMT-Based Approach to Coverability Analysis*. In: *CAV*, *Lecture Notes in Computer Science* 8559, Springer, pp. 603–619, doi:10.1007/978-3-319-08867-9_40.

[16] Javier Esparza & Philipp J. Meyer (2015): *An SMT-based Approach to Fair Termination Analysis*. In: *FMCAD*, IEEE, pp. 49–56, doi:10.1109/FMCAD.2015.7542252.

[17] Javier Esparza, Mikhail Raskin & Christoph Welzel (2021): *Abduction of trap invariants in parameterized systems*. Available at `https://arxiv.org/abs/2108.09101`.

[18] Javier Esparza, Mikhail Raskin & Christoph Welzel (2021): *heron, git repository*. `https://gitlab.lrz.de/i7/heron`.

[19] Javier Esparza, Mikhail A. Raskin & Christoph Welzel (2021): *Computing Parameterized Invariants of Parameterized Petri Nets*. In: *Petri Nets*, *Lecture Notes in Computer Science* 12734, Springer, pp. 141–163, doi:10.1007/978-3-030-76983-3_8.

[20] Alain Finkel & Philippe Schnoebelen (2001): *Well-structured transition systems everywhere!* *Theor. Comput. Sci.* 256(1-2), pp. 63–92, doi:10.1016/S0304-3975(00)00102-X.

[21] Pierre Ganty & Rupak Majumdar (2012): *Algorithmic verification of asynchronous programs*. *ACM Trans. Program. Lang. Syst.* 34(1), p. 6, doi:10.1145/2160910.2160915.

[22] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub & Marius Schneider (2011): *Potassco: The Potsdam Answer Set Solving Collection*. *AI Commun.* 24(2), pp. 107–124, doi:10.3233/AIC-2011-0491.

[23] Steven M. German & A. Prasad Sistla (1992): *Reasoning about systems with many processes*. *Journal of the ACM (JACM)* 39(3), pp. 675–735, doi:10.1145/146637.146681.

[24] Silvio Ghilardi & Silvio Ranise (2010): *MCMT: A Model Checker Modulo Theories*. In: *IJCAR*, *Lecture Notes in Computer Science* 6173, Springer, pp. 22–29, doi:10.1007/978-3-642-14203-1_3.

[25] Bernhard Gleiss, Laura Kovács & Lena Schnedlitz (2019): *Interactive Visualization of Saturation Attempts in Vampire*. In: *IFM*, *Lecture Notes in Computer Science* 11918, Springer, pp. 504–513, doi:10.1007/978-3-030-34968-4_28.

[26] Maurice Herlihy & Nir Shavit (2008): *The art of multiprocessor programming*. Morgan Kaufmann.

[27] Emmanuel Jeandel (2010): *The periodic domino problem revisited*. *Theor. Comput. Sci.* 411(44-46), pp. 4010–4016, doi:10.1016/j.tcs.2010.08.017.

[28] Donald E. Knuth (1966): *Additional comments on a problem in concurrent programming control*. *Commun. ACM* 9(5), pp. 321–322, doi:10.1145/355592.365595.

[29] Laura Kovács & Andrei Voronkov (2013): *First-Order Theorem Proving and Vampire*. In: *CAV*, *Lecture Notes in Computer Science* 8044, Springer, pp. 1–35, doi:10.1007/978-3-642-39799-8_1.

[30] Nancy A. Lynch (1996): *Distributed Algorithms*. Morgan Kaufmann.

[31] Christian Müller, Helmut Seidl & Eugen Zalinescu (2018): *Inductive Invariants for Noninterference in Multi-agent Workflows*. In: *CSF*, IEEE Computer Society, pp. 247–261, doi:10.1109/CSF.2018.00025.

[32] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv & Sharon Shoham (2016): *Ivy: safety verification by interactive generalization*. In: *PLDI*, ACM, pp. 614–630, doi:10.1145/2908080.2908118.

[33] Giles Reger (2016): *Better Proof Output for Vampire*. In: *Vampire@IJCAR*, *EPiC Series in Computing* 44, EasyChair, pp. 46–60, doi:10.29007/5dmz.

[34] G. Sutcliffe (2017): *The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0*. *Journal of Automated Reasoning* 59(4), pp. 483–502, doi:10.1007/s10817-017-9407-7.

[35] Christoph Welzel, Javier Esparza & Mikhail Raskin (2020): *heron, software artifact*. `https://doi.org/10.5281/zenodo.5068849`, doi:10.5281/zenodo.5068849.