Formal Methods in System Design
https://doi.org/10.1007/s10703-023-00439-6

Thread-modular counter abstraction: automated safety and termination proofs of parameterized software by reduction to sequential program verification

Thomas Pani[1] · Georg Weissenbacher[2] · Florian Zuleger[2]

Received: 16 March 2021 / Accepted: 13 August 2023

Abstract
Parameterized programs are composed of an arbitrary number of concurrent, infinite-state threads. Automated safety and liveness proofs of such parameterized software are hard; state-of-the-art methods for their formal verification rely on intricate abstractions and complicated proof techniques that impede automation. In this paper, we introduce *thread-modular counter abstraction* (TMCA), a lean new abstraction technique to replace the existing heavy proof machinery. TMCA is a structured abstraction framework built from a novel combination of *counter abstraction*, *thread-modular reasoning*, and *predicate abstraction*. Its major strength lies in reducing the parameterized verification problem to the sequential setting, for which powerful proof procedures, efficient heuristics, and effective automated tools have been developed over the past decades. In this work, we first introduce the TMCA abstraction paradigm, then present a fully automated method for parameterized safety proofs, and finally discuss its application to automated termination and liveness proofs of parameterized software.

Keywords Parameterized program · Parameterized safety · Parameterized termination · Counter abstraction · Thread-modular reasoning · Predicate abstraction

1 Introduction

In this paper, we present a novel abstraction method for so-called *parameterized programs*, i.e., infinite-state programs that are executed by an unbounded number of concurrent threads. Our abstraction allows us to automatically prove safety and termination of such

✉ Thomas Pani
thomas@thpani.net

Georg Weissenbacher
georg.weissenbacher@tuwien.ac.at

Florian Zuleger
florian.zuleger@tuwien.ac.at

[1] Informal Systems and TU Wien, Vienna, Austria

[2] TU Wien, Vienna, Austria

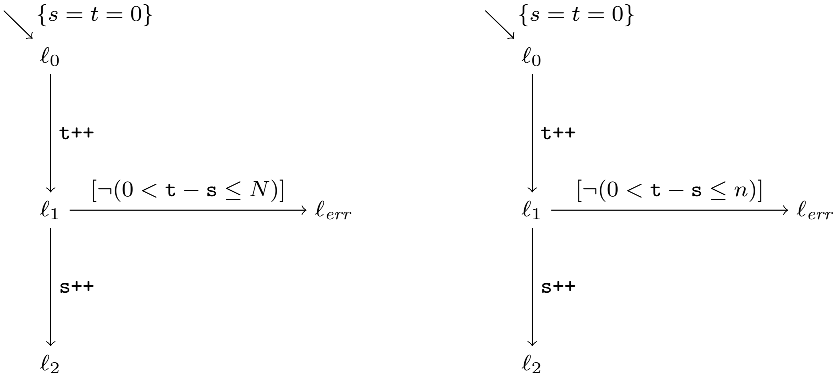
Published online: 06 October 2023

programs. In this section, we describe the main obstacles to finding computational safety and liveness proofs for parameterized programs.

$$\{s = t = 0\}$$

```
t++;
assert(0 < t − s ≤ N);
s++;
```

(a) Source code of program template $T[N]$.



(b) Control-flow graph of template $T[N]$.

(c) Program $P = T[N/n]$.

$$\{s = t = 0\}$$

$$\underbrace{\begin{array}{l}\texttt{t++;}\\ \texttt{assert}(0 < \texttt{t} - \texttt{s} \le n)\texttt{;}\\ \texttt{s++;}\end{array} \;\Bigg\|\; \quad \ldots \quad \;\Bigg\|\; \begin{array}{l}\texttt{t++;}\\ \texttt{assert}(0 < \texttt{t} - \texttt{s} \le n)\texttt{;}\\ \texttt{s++;}\end{array}}_{n \text{ copies of } P}$$

(d) Parameterized program $P(n) = P_1 \parallel \cdots \parallel P_n$.

$$\{s = t = 0\}$$

$$\underbrace{\begin{array}{l}\texttt{t++;}\\ \texttt{assert}(0 < \texttt{t} - \texttt{s} \le 3)\texttt{;}\\ \texttt{s++;}\end{array} \;\Bigg\|\; \begin{array}{l}\texttt{t++;}\\ \texttt{assert}(0 < \texttt{t} - \texttt{s} \le 3)\texttt{;}\\ \texttt{s++;}\end{array} \;\Bigg\|\; \begin{array}{l}\texttt{t++;}\\ \texttt{assert}(0 < \texttt{t} - \texttt{s} \le 3)\texttt{;}\\ \texttt{s++;}\end{array}}_{3 \text{ copies of } P}$$

(e) Instance $P(3) = P_1 \parallel P_2 \parallel P_3$ of size $n = 3$ of the parameterized program $P(n)$.

Fig. 1 Safety running example. Adapted from the introductory example of [25] by extending the assertion with an upper bounds check $t - s \le N$ on the parameter

1.1 Safety proofs of parameterized programs

Running example (Safety) Consider the *program template* $T[N]$ over global variables `s` and `t` and parameter N whose source code and control-flow graph are shown in Fig. 1a, b, respectively.[1] Assume that T is executed by an arbitrary number of n threads, where each thread runs the program $P = T[N/n]$ obtained by replacing parameter N by the number of threads $n > 0$ in program template T (Fig. 1c). We write $P(n) = P_1 \parallel \cdots \parallel P_n$ for this *parameterized program* (Fig. 1d). Figure 1e shows the *instance* $P(3)$ of parameterized program $P(n)$ for size $n = 3$. In this paper, we show how to automatically prove that the error location ℓ_{err} is unreachable for all threads from an initial state of, e.g., $s = t = 0$ for all system sizes $n > 0$. That is, we prove safety of the *infinite family* $\{P(1), P(2), \dots\}$ of instances induced by the parameterized program $P(n)$.

1.1.1 Ingredients of a safety proof

Despite the seemingly simple structure of the program, automatically constructing such a safety proof is hard: it needs to relate the unboundedly many local states of all threads, the arbitrary number of threads n, and the global variables `s` and `t` in a meaningful way. For example, note that the value of global variable `t` equals the number of threads at either control location ℓ_1, ℓ_2, or ℓ_{err}. Similarly, the value of `s` equals the number of threads at control location ℓ_2. Indeed, the safety proof of the assertion's lower bound $0 < t - s$ requires finding the invariant

$$\text{(number of threads in control-state } \ell_1) \leq t - s. \tag{1}$$

As a further complication, the assertion not only refers to variables, but also to the parameter n.

1.1.2 Role of our abstraction

Our abstraction TMCA provides a concise way to find proofs relating the unboundedly many local states of all threads, the parameter n, and the global variables of the parameterized program. Additionally, TMCA does not prescribe a specific way of finding these arguments: It yields its abstraction of the parameterized program as a single sequential program that is amenable (in theory) to all existing software verification approaches for sequential software. Thus, TMCA allows us to verify the *infinitely-many instances* of the parameterized program *all at once* by checking a *single thread-modular summary program* using an off-the-shelf safety prover for *sequential software*. In addition, we benefit from the wealth of knowledge and automation for sequential software verification built over the past decades [22, 45].

[1] This slightly abstracted version of a ticket lock is adapted from the introductory example of [25]. We extend their version with an upper bounds check $t - s \leq N$ on the parameter N. This allows us to bound $t - s$ from above by the number of actual concurrent threads n whereas [25] only check the lower bound $0 < t - s$, i.e, that $s < t$.

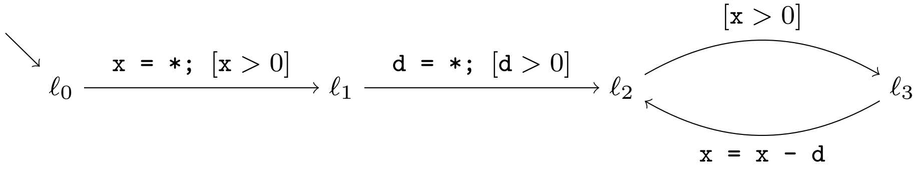


Fig. 2 Termination running example. This is the introductory example of [27]

1.2 Termination proofs of parameterized programs

In addition to safety properties, we consider *termination*, a foundational liveness property and important stepping stone towards the verification of general liveness properties [15, 71].

Running example (Termination) Consider the program $\mathcal{P}$ shown in Fig. 2. First, variable x and then variable d are assigned positive integer values ($\ell_0 \rightarrow \ell_1$ and $\ell_1 \rightarrow \ell_2$, respectively). The subsequent loop subtracts the value of d from x ($\ell_3 \rightarrow \ell_2$) until x becomes non-positive ($\ell_2 \rightarrow \ell_3$).

Our goal is to check that—starting from an arbitrary initial program state—the parameterized program $\mathcal{P}(n)$ terminates, i.e., that all its instances $\mathcal{P}(1), \mathcal{P}(2), \ldots$ take only finitely-many program steps.

1.2.1 Ingredients of a termination argument

We consider the arguments that lead to a successful termination proof of our example program $\mathcal{P}(n)$.

Sequential termination It is easy to see that a single copy of program $\mathcal{P}$ terminates: Because the value of variable d is always positive, the value of x strictly decreases in the loop $\ell_2 \rightarrow \ell_3 \rightarrow \ell_2 \rightarrow \ldots$. Eventually, the value of x becomes non-positive. The loop's execution is guarded by the assume statement [x > 0], i.e., once the value of x falls below one, the program stops executing and terminates.

Unbounded interference However, the induced parameterized program $\mathcal{P}(n)$ of $n > 0$ threads executing the same program template is more evolved: Consider a reference thread currently executing the loop $\ell_2 \rightarrow \ell_3 \rightarrow \ell_2 \rightarrow \ldots$. An unbounded number of environment threads can execute transition $\ell_0 \rightarrow \ell_1$ to reset the value of x an unbounded number of times. The value of x may never become non-positive, causing the reference thread's loop to diverge.

Fixed-size terminiation The termination argument for a concrete instance (i.e., for a fixed system size n), is that this reset of x happens at most a fixed number (viz. $n - 1$) of times while some thread is within the loop. Still, to prove termination of the *infinite family* of fixed-size instances $\{\mathcal{P}(1), \mathcal{P}(2), \ldots\}$ induced by the parameterized program $\mathcal{P}(n)$, a naive approach would need to repeat this argument for each of the infinite number of system sizes $n > 0$. In addition, such a proof needs to keep track of the progress of unboundedly-many threads, and thus may need to refer to their unboundedly-many local states.

Need for abstraction It is clear that we need a stronger argument that allows us to treat all $n > 0$ instances of the parameterized program at the same time. Indeed, our TMCA abstraction from Sect. 6 allows us to do this by folding the infinitely-many stances into a single thread-modular summary program.

1.2.2 Role of our abstraction

Our abstraction TMCA provides a concise way to capture the finiteness argument from above while preserving the behavior of all infinitely-many instances of the parameterized program. As for safety properties in Sect. 6, TMCA yields a thread-modular summary program that can be verified in a *single run* of an off-the-shelf termination prover for *sequential software*, and implies termination of *all infinitely-many instances* of the parameterized program *at once*. *Role of our abstraction.* Our abstraction TMCA provides a concise way to capture the finiteness argument from above while preserving the behavior of all infinitely-many instances of the parameterized program. Again, TMCA yields a thread-modular summary program that

- can be verified in a *single run* of an off-the-shelf termination prover for *sequential software*, and
- implies termination of *all infinitely-many instances* of the parameterized program *at once*.

1.3 Parameterized programs: two dimensions of infinity

Parameterized programs—like the ones above—induce an infinite family of concurrent programs, one for each instantiation of the parameter n. Together, this family of concurrent programs exhibits the following *Dimensions of Infinity* that any automated procedure has to deal with:

(I) **Unbounded replication of local state** The program template's control structure and local variables are replicated for each of the unboundedly many threads.

(II) **Infinite data domain** As with sequential software, we use a standard approach in software verification, where finite domains (e.g., machine integers) are abstracted to idealized infinite domains (e.g., the mathematical integers). [45]

Relation to sequential program verification The addition of Dimension (I) makes parameterized program verification significantly harder than the sequential case. It is not sufficient to merely find a suitable abstraction for this added dimension; as we demonstrated on the examples above, the complex interaction between Dimensions (I) and (II) makes parameterized program verification highly complex and requires meticulous abstraction design to capture the interactions between Dimensions (I) and (II) in a meaningful way.

Relation to sequential program verification State-of-the-art methods rely on heavy proof machinery to tackle these Dimensions of Infinity (cf. Related Work in Sect. 2). In contrast, our method is a novel combination of well-known techniques. Significantly improving the state of the art, we build a powerful and cleanly structured two-step abstraction framework. Our method is fully automated and treats the infinity dimensions in dedicated abstraction layers, thus providing a clear interface for abstracting their interactions.

Structure of our approach The first step of our method, *thread-modular counter abstraction* (TMCA), deals with Infinity Dimension (I) and is inspired by the well-known techniques counter abstraction [63] and thread-modular reasoning [29, 40, 47]. TMCA uses symmetry reduction to track the number of threads in a specific local state, encodes this information in the (already infinite) data domain (i.e, Infinity Dimension (II)), and abstracts the unbounded local state into a (local-)stateless thread-modular summary. TMCA models are sequential programs that can be checked using off-the-shelf software verifiers to handle Infinity Dimension (II). However, our experiments show that state-of-the-art safety techniques tend to diverge on our abstractions. We thus tackle Infinity Dimension (II) by presenting a *novel predicate refinement heuristic* for predicate abstraction [10, 35].

2 Related work

There exists extensive research on the automated verification of *parameterized systems*, i.e., the unbounded replication of *finite-state* components. The survey in [74] gives an extensive overview. In contrast, we are interested in the verification of *parameterized programs*, where already the individual components are *infinite-state*. Several works discuss their verification, among them approaches orthogonal to ours such as *cutoff detection* [48, 50], semi-automatic *deductive techniques* [54], or those based on *small model properties* [9, 62]. In the following, we discuss the works most closely related to ours.

2.1 Existing methods for safety verification

Gurfinkel et al. [38] generalizes the Owicki–Gries approach [56] to find universally quantified inductive invariants for parameterized systems. They encode k asynchronous processes as a set of Constrained Horn Clauses (CHCs) and use a solver to check for the existence of an invariant with k universally quantified variables. Syntactic limitations of the system and the invariant candidates guarantee that the Herbrand universe these quantified variables induce is finite and can be instantiated, which yields a decision procedure for several decidable fragments (such as Petri nets).

Model checking modulo theories (MCMT) [32, 33] builds on verification techniques for *well-structured transition systems* (WSTS) [2, 28] to model-check a subclass of array-based systems (namely, those expressible in *well-quasi-ordered* theories). Follow-up work in [5] presents practical improvements to MCMT by combining it with interpolation-based abstraction. While MCMT also includes some heuristics for predicate-abstraction, the main focus of MCMT is a symbolic approach to backward-reachability based on WSTS; in particular, WSTS support the direct analysis of infinite state systems and can guarantee the termination of the analysis under certain conditions. In contrast, we don’t build on WSTS and propose a specific abstraction technique for reducing the safety (and liveness) verification problem of parameterized system to a finite-state system.

Ganjei et al. [30, 31] prove parameterized program safety by combining two nested CEGAR loops: Their method applies *symmetric predicate abstraction* [21], a specialization of predicate abstraction for symmetric concurrent programs, to obtain a program template's finite-state abstraction as a boolean program. The method then uses counter abstraction to encode the parallel composition of n copies of the boolean program into a monotonic counter machine (essentially a vector addition system, i.e., more threads lead to more behavior). Since some wide-spread synchronization constructs have *non-monotonic* behavior, these tests are lost in the monotonic abstraction[2]. The authors strengthen their abstraction using a thread-modular analysis and check the resulting, now non-monotonic counter machine with the inner CEGAR loop running *constrained monotonic abstraction* [1], again abstracting the non-monotonic system into a monotonic one for which state reachability is decidable.

Kaiser et al. [49] present another combination of monotonic abstraction nested inside a specialized predicate abstraction. They introduce a symbolic representation for tracking inter-thread predicates, extending those of [21]. The resulting system is again non-monotonic and the authors force monotonicity as above.

Following a different approach, Farzan et al. [25] introduce *control flow nets*, a hybrid of Petri nets and control flow graphs, as their program model. The proof procedure alternates between synthesizing a candidate *counting automaton* (a kind of restricted counter machine) and checking language inclusion with the underlying control flow net. While this inclusion check is decidable, it has high computational complexity and no implementation is given. In addition, the Petri net program model has several shortcomings. It is unclear which parameterized verification problems can be modeled by the suggested formalism. For example, even the authors state a program where "it does not seem possible to encode the verification problem for mutual exclusion by a control flow net" [25]. Moreover, it is unclear how to express the additional upper-bounds check on n added to our running example (Fig. 1c) given that the parameter is not symbolically represented in the control flow net.

In a later paper, Farzan et al. [26] extend the language-theoretic approach to program correctness for sequential programs by Heizmann et al. [39] to the parameterized setting. Their algorithm generalizes single infeasible counter-example traces to sets of infeasible traces that are recorded in the newly-introduced *predicate automata*. The paper focuses on the formal development but leaves important implementation questions unanswered. In particular, the efficacy of this approach is limited by (i) only considering a single infeasible counter-example trace at a time, and (ii) by the ability to heuristically divine a generalization to a set of infeasible traces that is suitable for the overall program. On the engineering side, the verification algorithm has to iteratively perform emptiness checks on the constructed predicate automata. This itself is an undecidable problem and may be intractable even in practical settings.

[2] Synchronization mechanisms such as the *dynamic barriers* considered by Ganjei et al. [30, 31] test the number of threads in a specific state. In essence, their counter abstraction would then have to encode a counter machine with zero tests, making state reachability checking undecidable.

2.2 Existing methods for termination proving

Cook et al. [15] prove *lock-freedom*, a liveness property of pointer-manipulating programs [41], via reduction to termination. They give particular focus to the development of an iterative thread-modular proof rule suited for pointer programs and this specific liveness property. Their construction requires a termination check in each iteration, which is often costly and unnecessary, as we demonstrate in Sect. 9.

In our own previous work [59], we introduce an iterative thread-modular method for performing *resource bound analysis* [37], which is equivalent to the synthesis of *bounded* liveness properties [61]. Like Cook et al. [15], this method incrementally refines its environment abstraction and requires a costly bound analysis step in each iteration.

Farzan et al. [27] present a liveness-to-safety reduction by introducing *well-founded proof spaces* and *quantified predicate automata* (QPAs), both new formalisms for computing termination arguments for parameterized programs. QPAs extend the *predicate automata* of [26] (see above), themselves a relatively new addition to the software verification toolbelt. Similar to their previous work, the authors only consider a single terminating trace at a time and rely on checking QPA emptiness, itself an undecidable problem. A possible solution to the latter problem is only sketched in the paper; consequently, it is unclear whether an effective implementation of this approach is feasible.

Padon et al. [58] reduce the liveness verification of parameterized programs to safety of infinite-state systems expressed in pure first-order logic. They forego synthesis of ranking functions altogether and focus on an abstract semantics for which acyclicity proves termination. In their paper, the authors apply liveness-to-safety reduction manually, and follow it with an interactive safety proof; i.e., the verification is mechanized, but not push-button automatic.

2.3 Our approach

In summary, state-of-the-art methods rely on tightly coupled, specialized abstractions and heavy, non-standard proof machinery. Many times, implementation questions are unclear and the possibility of automation is questionable. However, our experiments show that many practical examples can be proven in a more straight-forward way: We replace the heavy machinery of previous work with a *clean, two-step abstraction framework built from a novel combination of well-known techniques*, thus significantly improving the state of the art.

In particular, we start from a *standard program model* by encoding program templates as transition systems. To these, our method first applies a novel thread-modular counter abstraction adapted to infinite-state systems that tracks and projects away the unboundedly replicated local state. In the subsequent step, we apply *standard predicate abstraction* [10, 35] (for safety) and *standard transition predicate abstraction* [16] (for termination) to deal with the infinite data domain. The discovery of counting arguments is left entirely to the abstraction refinement phase. This reduces reasoning to the sequential setting, which has seen a wealth of research into both theoretical and practical aspects over the past decades [22, 45]. We show in Sects. 8 and 9 that this straight-forward method is powerful enough for many examples from the literature. In addition, our two-step abstraction follows a clean design by applying the *separation of concerns* design principle: each Dimension of Infinity (cf. Sect. 1.3) is dealt with in a dedicated component. While our upfront thread-modular abstraction may be too coarse in some cases, it could be strengthened by an outer refinement loop, again running *predicate abstraction*. This additional refinement step is beyond the scope of this work; we sketch it in Sect. 10 and leave its detailed investigation for future work.

3 Contributions

We introduce a novel framework for parameterized software verification. Its advantages over state-of-the-art methods lie in its clean design and simplicity, while being powerful enough to tackle the benchmarks of previous work—in the case of safety even a superset. In particular, we make the following contributions:

1. Our framework is a novel layered proof system of well-understood and pluggable components. The power of our method stems from adapting, combining, and extending established methods without introducing complicated new proof machinery or non-standard concepts for both safety properties (Sects. 6 and 7) and termination (Sect. 9). To our knowledge, we are the first to suggest this combination of techniques for safety and termination proofs of parameterized programs. In particular, we contribute the following technical advancements:

    (a) We adapt counter abstraction to infinite-state systems by introducing auxiliary state to track the number of threads in a specific local state (Sect. 6). To our knowledge we are the first to propose such a counter abstraction and to apply it to parameterized programs.

    (b) We reduce reasoning about parameterized programs to the sequential setting. If the abstracted sequential program is safe or terminates, then the original parameterized program is also safe (Sect. 6) or terminates (Sect. 9), respectively. This facilitates the reuse of existing sequential software verifiers for parameterized verification.

    (c) Predicate abstraction with standard predicate selection heuristics diverges on our abstract models (Sect. 8). We present novel predicate selection heuristics to guide a CEGAR loop in the presence of these counter-abstracted summaries (Sect. 7).

2. We implement our safety method based on constrained Horn clauses (CHCs) and demonstrate its efficacy on a combined benchmark set from various sources (Sect. 8). To our knowledge, our technique is the only automated method that has been demonstrated to successfully solve this benchmark set.
3. We present a case study demonstrating how to extend these results to termination (Sect. 9). In addition, we demonstrate the efficacy of this method by automatically proving termination of a standard example from the literature using an existing termination prover for sequential programs (Sect. 9). To our knowledge, this constitutes the first actual implementation of an automated termination proof for this benchmark.
4. The individual components of our framework lend themselves to tweaking and adaptation. We discuss avenues for further research, both on the theoretical side (e.g., by providing new heuristics or refinement methods) and on the practical side (e.g., through new and improved backend solvers) (Sect. 10).

Extended version This is an extended version of the conference paper [60] that appeared at FMCAD 2020. The conference version solely considered safety analysis. In this extended version, besides making the material more accessible through additional explanations and discussions, we extend our framework to termination proving. This additional content is

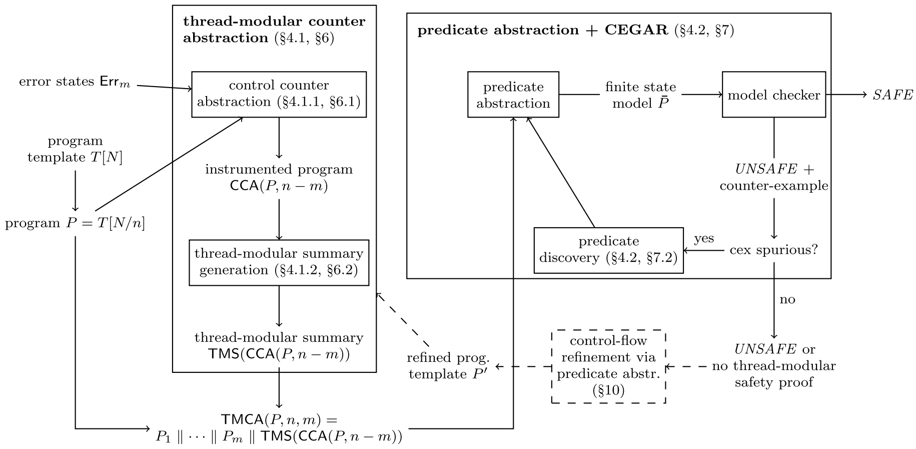

thread-modular counter abstraction (§4.1, §6)
error states Err_m
control counter abstraction (§4.1.1, §6.1)
program template $T[N]$
program $P = T[N/n]$
instrumented program $\mathsf{CCA}(P, n-m)$
thread-modular summary generation (§4.1.2, §6.2)
thread-modular summary $\mathsf{TMS}(\mathsf{CCA}(P, n-m))$
$\mathsf{TMCA}(P, n, m) =$
$P_1 \parallel \cdots \parallel P_m \parallel \mathsf{TMS}(\mathsf{CCA}(P, n-m))$
predicate abstraction + CEGAR (§4.2, §7)
predicate abstraction
finite state model $\bar{P}$
model checker
SAFE
UNSAFE + counter-example
cex spurious?
yes
predicate discovery (§4.2, §7.2)
no
UNSAFE or no thread-modular safety proof
control-flow refinement via predicate abstr. (§10)
refined prog. template P'

Fig. 3 Overall structure of our method for safety verification. Dashed parts are beyond the scope of this work and sketched in Sect. 10. In parentheses, we reference first the corresponding section of our motivating example (Sect. 4) and second the detailed technical exposition (Sects. 6 and 7)

naturally spread throughout the paper, with the major additions and results presented in Sects. 6 and 9.

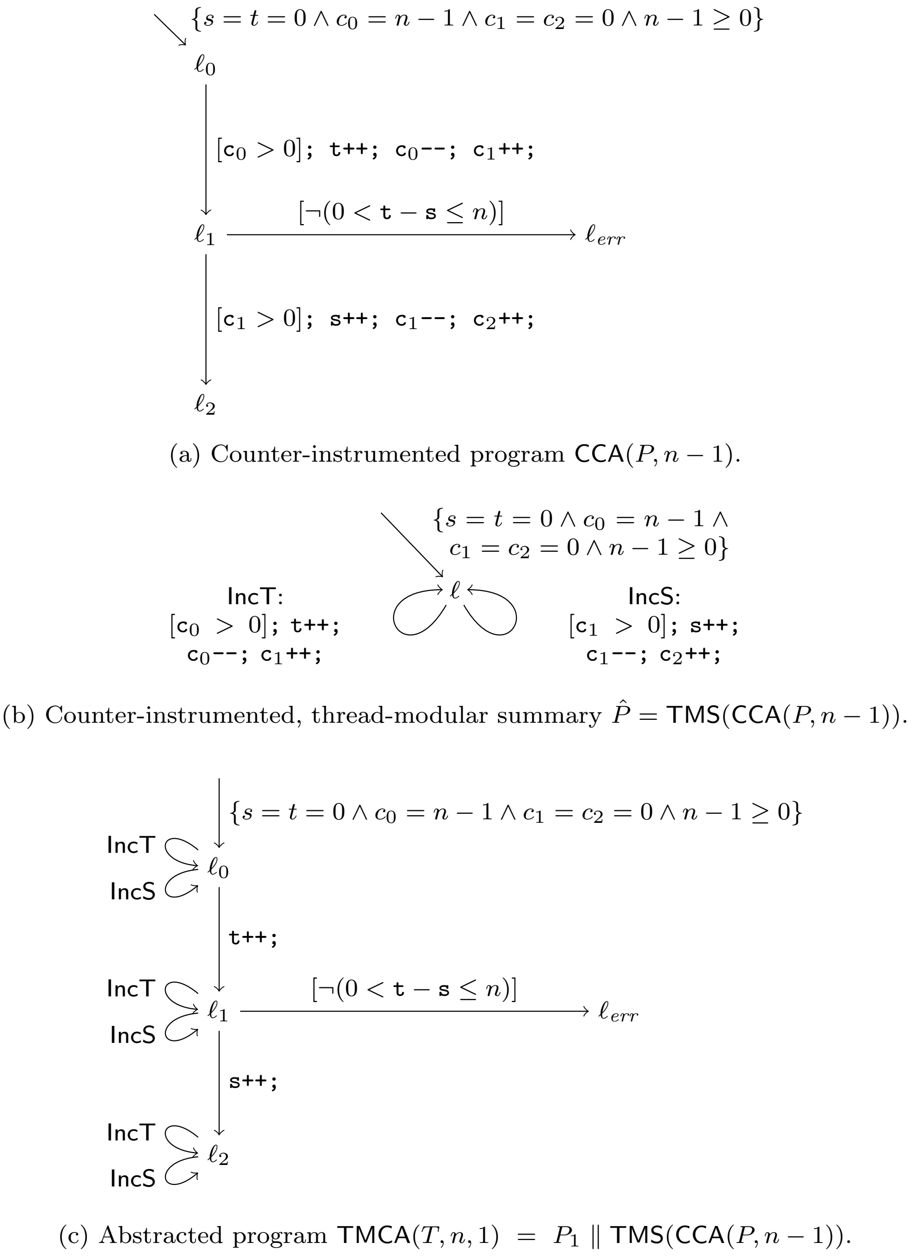
4 Motivating example: safety

In this section, we give an overview of our approach for safety analysis on our running example (we continue to develop our method for termination in Sect. 9, referencing many of the results for safety). This will help to distinguish our method from existing, state-of-the-art methods, which we already discussed in Sect. 2.

Starting from a program template $T[N]$, our method constructs the base copy of a parameterized program by replacing parameter N with the total number of threads n to obtain program $P = T[N/n]$. We are then interested in the safety of the parameterized program $P(n)$ induced by P for all $n > 0$.

4.1 Overall idea: thread-modular abstraction

To tackle the unbounded replication of parameterized programs (cf. Sect. 4), our method keeps one thread concrete and computes an abstraction of the $n-1$ other threads. In line with thread-modular reasoning terminology [29, 40, 47], we call these $n-1$ threads *environment threads* or just *the environment*. The following sections demonstrate how we compute this abstraction on our introductory example. Figure 3 shows the overall structure of our method.


$\{s = t = 0 \wedge c_0 = n - 1 \wedge c_1 = c_2 = 0 \wedge n - 1 \geq 0\}$
ℓ_0
$[\mathtt{c}_0 > 0]$; t++; $\mathtt{c}_0$--; $\mathtt{c}_1$++;
ℓ_1
$[\neg(0 < \mathtt{t} - \mathtt{s} \leq n)]$
ℓ_{err}
$[\mathtt{c}_1 > 0]$; s++; $\mathtt{c}_1$--; $\mathtt{c}_2$++;
ℓ_2

(a) Counter-instrumented program $\mathsf{CCA}(P, n-1)$.

$\{s = t = 0 \wedge c_0 = n - 1 \wedge$
$c_1 = c_2 = 0 \wedge n - 1 \geq 0\}$
ℓ
IncT:
$[\mathtt{c}_0 > 0]$; t++;
$\mathtt{c}_0$--; $\mathtt{c}_1$++;
IncS:
$[\mathtt{c}_1 > 0]$; s++;
$\mathtt{c}_1$--; $\mathtt{c}_2$++;

(b) Counter-instrumented, thread-modular summary $\hat{P} = \mathsf{TMS}(\mathsf{CCA}(P, n-1))$.

$\{s = t = 0 \wedge c_0 = n - 1 \wedge c_1 = c_2 = 0 \wedge n - 1 \geq 0\}$
IncT
IncS
ℓ_0
t++;
IncT
IncS
ℓ_1
$[\neg(0 < \mathtt{t} - \mathtt{s} \leq n)]$
ℓ_{err}
s++;
IncT
IncS
ℓ_2

(c) Abstracted program $\mathsf{TMCA}(T, n, 1) \;=\; P_1 \parallel \mathsf{TMS}(\mathsf{CCA}(P, n-1))$.

Fig. 4 Running example illustrating our thread-modular abstraction TMCA. This demonstrates our abstraction on the introductory example for safety in Fig. 1

4.1.1 Counter instrumentation

Our thread-modular abstraction below keeps one thread concrete and abstracts the remaining $n-1$ environment threads. To track the local state of these $n-1$ environment threads, our method instruments program P with additional global counter variables. This introduction of auxiliary state serves to retain some information about the local state of the environment threads in the subsequent abstraction step.

Running example (Safety) In our motivating example (Fig. 1), each thread's local state is given entirely by the valuation of its program counter, which ranges over the finite domain of program locations $\{\ell_0, \ell_1, \ell_2\}$. Our method introduces fresh global variables $\{\mathsf{c}_0, \mathsf{c}_1, \mathsf{c}_2\}$ and instruments the program such that variable c_i tracks the number of threads at location ℓ_i. The resulting instrumented program $\mathsf{CCA}(P, n-1)$ is shown in Fig. 4a.

4.1.2 Thread-modular summary generation

In this step, our method uses thread-modular reasoning to project away the unboundedly many local variables of the $n-1$ environment threads. Our method generates a thread-modular summary $\hat{P}$ of the instrumented program $\mathsf{CCA}(P, n-1)$, such that $\hat{P}$ over-approximates the reachable global state space of the environment threads for all $n > 0$.

Running example (Safety) In our example, the only local variable of

$$\mathsf{CCA}(P, n-1) \tag{2}$$

(Fig. 4a) is the program counter. By projecting it away, we obtain

$$\hat{P} = \mathsf{TMS}(\mathsf{CCA}(P, n-1)) \tag{3}$$

as the thread-modular summary in Fig. 4b: Abstract transition IncT corresponds to transition $\ell_0 \rightarrow \ell_1$, and abstract transition IncS corresponds to transition $\ell_1 \rightarrow \ell_2$.

It is easy to see that from its initial state

$$\{s = t = 0 \wedge c_0 = n - 1 \wedge c_1 = c_2 = 0 \wedge n - 1 \geq 0\}, \tag{4}$$

the summary $\hat{P}$ over-approximates the globally visible behavior of $n-1$ environment threads for all $n > 0$. Thus, instead of analyzing the parameterized program $P(n)$ or its infinitely many instances $\{P(1), P(2), \ldots\}$, we instead consider its over-approximation

$$\mathsf{TMCA}(T, n, 1) = P_1 \parallel \hat{P} \tag{5}$$

shown in Fig. 4c, where summary $\hat{P}$ over-approximates the behavior of the unboundedly-many environment threads $P_2 \parallel \cdots \parallel P_n$.

4.2 Invariant generation (predicate abstraction + CEGAR)

The abstracted program $\mathsf{TMCA}(T, n, 1)$ from above is just a sequential program that could be checked by off-the-shelf software verifiers, e.g., based on predicate abstraction. Our experiments (Sect. 8) show that our abstraction already allows state-of-the-art methods to prove safety for *some* examples. However, due to the uncommon structure of our abstract models, standard predicate discovery heuristics often diverge. Again improving the state of the art, we thus introduce a novel predicate selection heuristic in Sect. 7.

Running example (Safety) For our abstracted example $\mathsf{TMCA}(T, n, 1)$ in Fig. 4c, this predicate selection procedure finds the following invariant at control location ℓ_1:

$$c_1 < t - s \leq n - c_0 \wedge c_0 \geq 0 \wedge c_1 \geq 0 \wedge c_2 \geq 0 \wedge n > 0 \wedge s \geq 0 \wedge t > 0 \tag{6}$$

Obviously, this implies that $0 < t - s \leq n$ and thus proves the error location ℓ_{err} unreachable.

5 Program model and parameterized safety

In the previous section, we demonstrated our approach for safety analysis on our running example. In this section, we start to formally develop the abstraction technique illustrated above by formalizing our program model and our problem statement for safety.

Definition 1 (*Program model*) Let $\mathbf{g} = (g_1, \ldots, g_k)$ and $\mathbf{l} = (l_1, \ldots, l_j)$ be disjoint tuples of *global* and *local program variables*. Let N be a *symbolic parameter*.

A *guarded command* $gc \in \mathsf{GC}$ over $\mathbf{l}$, $\mathbf{g}$, N has the form

$$gc \,:\; [cond] \mid v := e \mid gc_1; gc_2 \tag{7}$$

where $[cond]$ is an assume statement over $\mathbf{l}$, $\mathbf{g}$, N, and $v := e$ is an assignment of expression e over $\mathbf{l}$, $\mathbf{g}$, N to a local or global variable v. Expression $*$ denotes a non-deterministic integer value. We write $v(\mathbf{g}, \mathbf{l})$ for the valuation of global and local variables and omit its arguments wherever clear from the context. Further, we denote by $v\downharpoonright_{\mathbf{v}}$ the projection of valuation v to variables $\mathbf{v}$ and extend its definition to sets and relations in the usual, elementwise way. We denote by $v' \in [\![gc]\!](v)$ the (possibly non-deterministic) *effect* of a guarded command gc and write $\varphi(\mathbf{g}, \mathbf{g}', \mathbf{l}, \mathbf{l}')$ for its standard encoding as a formula over primed and unprimed variables.

A *program template* $T[N]$ over global and local variables $\mathbf{g}$ and $\mathbf{l}$ and a parameter N is a directed labeled graph $T[N] = (Loc, \delta, \ell_0, Init)$ where Loc is a finite set of *control locations*, $\ell_0 \in Loc$ is the *initial control location*, $\delta \subseteq Loc \times \mathsf{GC} \times Loc$ is a finite set of *transitions*, and $Init$ is a predicate over $\mathbf{g}$, $\mathbf{l}$, N describing the initial valuations of variables.

From template $T[N]$, we obtain *program* $P = T[N/n] = (Loc, \delta', \ell_0, Init')$ by replacing each occurrence of N in T (viz. in δ and $Init$) with the expression n. We call a pair (ℓ, v) of a control location $\ell \in Loc$ and a valuation $v(\mathbf{g}, \mathbf{l})$ a *program state*. We represent *runs* of P as interleaved sequences of states and transitions and write $(\ell_0, v_0) \xrightarrow{gc_0} (\ell_1, v_1) \xrightarrow{gc_1} \ldots$ such that v_0 satisfies $Init'$, and for all $i \geq 0$ we have that $(\ell_i, gc_i, \ell_{i+1}) \in \delta'$ and $v_{i+1} \in [\![gc_i]\!](v_i)$. Finally, let Reach (P) denote the *reachable states* of program P, i.e.,

$$\text{Reach}\,(P) \overset{\text{def}}{=} \{(\ell, v) \mid (\ell, v) \text{ occurs on a run of P}\}. \tag{8}$$

We overload the operator $\downharpoonleft_{\mathbf{v}}$ (projection to variables $\mathbf{v}$) to a set of program states Reach (P) by applying it element-wise to the program state's valuation component:

$$\text{Reach}\,(P)\downharpoonleft_{\mathbf{v}} \overset{\text{def}}{=} \{(\ell, v\downharpoonleft_{\mathbf{v}}) \mid (\ell, v) \text{ occurs on a run of } P\}. \tag{9}$$

Furthermore, let GReach (P) denote the *reachable global states* of P, i.e.,

$$\text{GReach}\,(P) \overset{\text{def}}{=} \{v\downharpoonleft_{\mathbf{g}} \mid (\ell, v) \text{ occurs on a run of P}\}. \tag{10}$$

We define the *transition relation* TR (P) of program P as

$$\text{TR}\,(P) \overset{\text{def}}{=} \{((\ell, v), (\ell', v')) \mid (\ell, v) \xrightarrow{gc} (\ell', v') \text{ appears on a run of } P\}, \tag{11}$$

and let GTR (P) denote the *transition relation* of program P *restricted to global states*, i.e.,

$$\text{GTR}\,(P) \overset{\text{def}}{=} \{(v\downharpoonleft_{\mathbf{g}}, v'\downharpoonleft_{\mathbf{g}}) \mid (\ell, v) \xrightarrow{gc} (\ell', v') \text{ appears on a run of P}\}. \tag{12}$$

We define the *interleaving* of two programs $P_1 = (Loc_1, \delta_1, \ell_{1,0}, Init_1)$ and $P_2 = (Loc_2, \delta_2, \ell_{2,0}, Init_2)$ over joint global variables $\mathbf{g}$ and disjoint local variables $\mathbf{l}_1$ and $\mathbf{l}_2$ as the program $P_1 \parallel P_2 = (Loc_1 \times Loc_2, \rho, (\ell_{1,0}, \ell_{2,0}), Init_1 \wedge Init_2)$ over global and local variables $\mathbf{g}$ and $\mathbf{l}_1 \cup \mathbf{l}_2$ where $((\ell_1, \ell_2), gc, (\ell'_1, \ell'_2)) \in \rho$ iff either $(\ell_1, gc, \ell'_1) \in \delta_1$ and $\ell'_2 = \ell_2$, or $(\ell_2, gc, \ell'_2) \in \delta_2$ and $\ell'_1 = \ell_1$. Let $P = (Loc, \delta, \ell_0, Init)$ be a program. For *thread identifiers* $i = 1, \ldots, k$ we obtain the *instantiation* P_i of program P by replacing each local variable l_j in P with its i-th copy $l_{j,i}$. We define the *k-times interleaving* P^k of P as the interleaving of the first k instantiations of P, i.e., $P^k = P_1 \parallel \cdots \parallel P_k$. Finally, a program template $T[N]$ induces a *parameterized program* $P(n) = (T[N/n])^n$, where parameter N has been replaced by the number of concurrent threads n and we construct the n-times interleaving of the resulting program.

Remark We note that we do not define a concrete syntax and semantics for the guards and commands, as the framework developed in this paper is not tied to a particular choice of guarded commands. In our implementation, the guards and commands are given by what is supported by the model checking backend.

Following [42, 44], we define *safety* of a parameterized program in the style of coverability:

Definition 2 (*Safety*) Let $T[N]$ be program template, and let $P(n)$ be its induced parameterized program over tuples of global and local variables $(\mathbf{g}, \mathbf{l}_1, \ldots, \mathbf{l}_n)$. Recall that a state of $P(n)$ has the form $((\ell_1, \ldots, \ell_n), v(\mathbf{g}, \mathbf{l}_1, \ldots, \mathbf{l}_n))$. We define *safety* relative to a generator set of error states Err_m of $(T[N/n])^m$ for a fixed $m \geq 0$. $P(n)$ is *safe* if and only if for all $n > 0$, no run of $P(n)$ reaches an error state from the system error states $\mathsf{Err}(n)$, where

$$\begin{aligned}\mathsf{Err}(n) \overset{\text{def}}{=} \{((\ell_1,\dots,\ell_n),v) \mid ((\ell_{i_1},\dots,\ell_{i_m}),v') \in \mathsf{Err}_m \text{ s.t.}\\ v'(\mathbf{g}) = v(\mathbf{g}), v'(\mathbf{l}_j) = v(\mathbf{l}_{i_j}) \text{ for } 0 \le j \le m \\ \text{and some } i_1,\dots,i_m \text{ s.t. } 1 \le i_1 < \dots < i_m \le n\},\end{aligned} \tag{13}$$

i.e., $P(n)$ is safe if and only if $\mathsf{Err}(n) \cap \text{Reach}\,(P(n)) = \emptyset$.

Intuitively, $P(n)$ is unsafe if it contains m pairwise distinct threads that reach an error state from Err_m while the remaining $n-m$ symmetric threads may take arbitrary control locations and local states.

Remark Note that for a concrete parameterized verification problem, m is a scalar value but n is universally quantified: Given a program template $T[N]$ and a generator set Err_m, our goal is to prove safety of the induced parameterized program $P(n)$, i.e., to show that reaching an error state from $\mathsf{Err}(n)$ is infeasible for all parameter instantiations $n > 0$.

To prove safety, our method follows a two-step process that we explain in the following two sections.

6 Tackling infinity dimension I: thread-modular counter abstraction

As outlined in Sect. 1.3, there are two main challenges in proving safety of a parameterized program $P(n)$: its *unboundedly replicated local state*, and the *infinite data domain*. The first step of our method, *thread-modular counter abstraction* (TMCA), tackles the first aspect. We target the second dimension, infinite data, in Sect. 7.

TMCA is inspired both by the work on *counter abstraction* [63] and *thread-modular reasoning* [29, 40, 47]. Starting from a program template $T[N]$, its induced parameterized program $P(n) = T[N/n]_1 \parallel \cdots \parallel T[N/n]_n$, and a generator set of error states Err_m, our goal is to construct a program abstraction $\hat{P}$ such that the single sequential program

$$\mathsf{TMCA}(T,n,m) \overset{\text{def}}{=} T[N/n]_1 \parallel \cdots \parallel T[N/n]_m \parallel \hat{P} \tag{14}$$

over-approximates the reachable state space of $P(n)$ for all $n > 0$, but has only finitely many control locations and variables. In the following, we explain both the counter abstraction and the thread-modular aspect of TMCA in further detail.

6.1 Control counter abstraction (CCA)

Background Counter abstraction [63] was introduced to abstract the parallel execution of *parameterized systems*, i.e., an unbounded number of *finite-state* processes: For each state, a counter is introduced to track how many processes reside in their respective copy of the state. Counter values are then projected onto a finite domain (e.g., [46, 74]) to obtain a finite-state system that is then model-checked. Counter abstraction has also been used for the verification of distributed systems [7] in order to tackle two dimensions of unboundedness: first the data received by each process is subjected to counter abstraction, then the resulting system of finite-state processes is counter-abstracted into a finite-state system. This idea has been adapted to parameterized software [30, 31, 49] by first predicate-abstracting the program template into a boolean program, and then counting the number of threads residing in one of the finitely many abstract states.

Our method In contrast, our method instruments counters as *auxiliary variables* [55, 56] into an *infinite-state* system: It is well-known that thread-modular reasoning is incomplete [8], but can be made more expressive by adding auxiliary state [54, 55]. Thus, in contrast to earlier work on counter abstraction, our goal is not to finitize the entire parameterized system, but to express the unboundedly replicated local state of a parameterized program $P(n)$ in the already infinite data domain. To this end, we first instrument the corresponding program P with fresh *counter variables*, one for each program location, that count the number of threads in (their copy of) the respective control state. We formalize this idea:

Definition 3 (*Auxiliary variable instrumentation* CCA) Let $P = (Loc, \delta, \ell_0, Init)$ be a program over global and local variables $\mathbf{g}$ and $\mathbf{l}$. We extend the set of global variables with a set of fresh auxiliary variables, one for each program location: for global variables $\mathbf{g} = (g_1, \dots, g_i)$ and control locations $Loc = \{\ell_0, \ell_1, \dots, \ell_j\}$, let the set of instrumented variables be

$$\mathbf{g}' = (g_1, \dots, g_i, c_0, c_1, \dots, c_j). \tag{15}$$

The *control counter-instrumented program* $\mathsf{CCA}(P, k) = (Loc, \delta', \ell_0, Init')$ is defined over the extended global variables $\mathbf{g}'$ and local variables $\mathbf{l}$ where the instrumented transition relation δ' is

$$\begin{aligned} &\ell_{src} \xrightarrow{gc'} \ell_{tgt} \in \delta' \quad \text{iff} \quad \ell_{src} \xrightarrow{gc} \ell_{tgt} \in \delta \quad \text{where} \\ &gc' \overset{\mathsf{def}}{=} [\mathsf{c}_{src} > 0];\, gc;\, \mathsf{c}_{src} := \mathsf{c}_{src} - 1;\, \mathsf{c}_{tgt} := \mathsf{c}_{tgt} + 1; \end{aligned} \tag{16}$$

and

$$Init' \overset{\mathsf{def}}{=} Init \wedge c_0 = k \wedge c_1 = \dots = c_j = 0 \wedge k \geq 0. \tag{17}$$

Proposition 1 *Let P be a program over global and local variables $\mathbf{g}$ and $\mathbf{l}$, and let P^k be its k-times interleaving over global variables $\mathbf{g}$ and local variables $\mathbf{l}_1 \cup \dots \cup \mathbf{l}_k$. Note that its control-counter instrumentation $\mathsf{CCA}(P, k)^k$ ranges over additional counters $\mathbf{c}$, i.e., over global variables $\mathbf{g} \cup \mathbf{c}$ and local variables $\mathbf{l}_1 \cup \dots \cup \mathbf{l}_k$. Up to the instrumented counter variables, the transition relation of $\mathsf{CCA}(P, k)^k$ is exactly the transition relation of P^k for all $k > 0$, i.e.,*

$$\mathrm{TR}\,(\mathsf{CCA}(P, k)^k)\!\restriction_{(\mathbf{g} \cup \mathbf{l}_1 \cup \dots \cup \mathbf{l}_k)} \;=\; \mathrm{TR}\,(P^k) \quad \text{for all } k > 0. \tag{18}$$

Proof sketch Every run of P^k can be extended to $\mathsf{CCA}(P, k)^k$: We can simply extend every state along the run of P^k to a state of $\mathsf{CCA}(P, k)^k$ by counting for every location of $Loc = \{\ell_0, \ell_1, \dots, \ell_j\}$ how many processes are at location ℓ_i and then assigning this value to the variable c_i.

On the other hand, every run of $\mathsf{CCA}(P, k)^k$ gives rise to a run of P^k by simply removing the instrumented variables.

The claim then follows because P^k and $\mathsf{CCA}(P,k)^k$ have the same runs (when projecting away the instrumented variables).

Corollary 1 *Up to the instrumented counter variables*, $\mathsf{CCA}(P,k)^k$ *has the same reachable states as* P^k *for all* $k > 0$, *i.e.*,

$$\text{Reach}\,(\mathsf{CCA}(P,k)^k)\downharpoonright_{(\mathbf{g}\cup\mathbf{l}_1\cup\cdots\cup\mathbf{l}_k)} \;=\; \text{Reach}\,(P^k) \quad \text{for all } k>0. \tag{19}$$

Remark Note that CCA's second argument k can be symbolic. We use this below to obtain a summary for an arbitrary number of threads n.

6.2 Thread-modular summary generation (TMS)

The parameterized program instrumented as outlined above still contains unboundedly many local variables. To tackle this second aspect of unboundedly replicated local state, our method computes a *thread-modular summary*.

Background. Originally conceived as an extension of Hoare logic to concurrency, *thread-modular reasoning* [29, 40, 47] picks one *reference thread* and models the interleaved steps of all other threads (the *environment*) in an *environment assumption*. This environment assumption is a binary relation over global program states and over-approximates the environment's transition relation.

Definition 4 (*Thread-modular summary*) Let $P = (Loc, \delta, \ell_0, Init)$ be a program. We call a program P' whose transition relation projected onto global states GTR (P') over-approximates the transition relation projected onto global states GTR (P) of P, i.e., for which we have GTR $(P) \subseteq$ GTR (P'), a *thread-modular summary* of P.

Our method. We compute thread-modular summaries by projecting away all local state (i.e., the control locations and valuations of local variables) from the program's transition relation[3]; in our framework this projection is simply expressed as existential quantification over the local variables[4]:

Definition 5 (*Thread-modular summary* TMS) Let $P = (Loc, \delta, \ell_0, Init)$ be a program over global and local variables $\mathbf{g}$ and $\mathbf{l}$. We define the *thread-modular summary* $\mathsf{TMS}(P) = (\{\ell\}, \delta', \ell, Init')$ over global variables $\mathbf{g}$ for a fresh program location $\ell \notin Loc$ where $Init' \overset{\text{def}}{=} \exists \mathbf{l}.Init$ and δ' is defined as

$$\ell \xrightarrow{\exists \mathbf{l},\mathbf{l}'\cdot\varphi(\mathbf{g},\mathbf{g}',\mathbf{l},\mathbf{l}')} \ell \in \delta' \quad \text{iff} \quad \ell_{src} \xrightarrow{\varphi(\mathbf{g},\mathbf{g}',\mathbf{l},\mathbf{l}')} \ell_{tgt} \in \delta. \tag{20}$$

[3] We choose this definition because it is sufficiently fine-grained for our safety benchmarks. In general, stronger notions of a thread-modular summary (e.g., restricting the transition relation to reachable states) can be adopted [40, 69, 70].

[4] In an implementation, one can either work with a set of guarded commands that supports quantifiers or use a quantifier elimination procedure to directly remove the quantified variables. In our prototype, we have implemented a simple quantifier elimination procedure, which was sufficient to deal with our benchmark set.

Proposition 2 *Let P be a program over global and local variables* $\mathbf{g}$ *and* $\mathbf{l}$. $\mathsf{TMS}(P)$ *is a thread-modular summary of P, i.e., its transition relation over-approximates the transition relation of P's k-times interleaving P^k when projected onto global states for all $k > 0$. Formally,*

$$\mathrm{GTR}\,(P^k) \;\subseteq\; \mathrm{GTR}\,(\mathsf{TMS}(P)) \quad \text{for all } k > 0. \tag{21}$$

Proof sketch The claim directly follows from Definition 5: Every run of $\mathrm{GTR}\,(P^k)$ is also a run of $\mathsf{TMS}(P)$ because we can always instantiated the existentially quantified variables in the run of $\mathsf{TMS}(P)$ with the values of the corresponding local variables in the run of $\mathrm{GTR}\,(P^k)$.

Corollary 2 $\mathsf{TMS}(P)$ *over-approximates the reachable global states of P's k-times interleaving P^k* for all $k > 0$, *i.e.*,

$$\mathrm{GReach}\,(P^k) \;\subseteq\; \mathrm{GReach}\,(\mathsf{TMS}(P)) \quad \text{for all } k > 0. \tag{22}$$

6.3 Combining CCA and TMS: thread-modular counter abstraction (TMCA)

The combination of control counter abstraction (Sect. 6.1) and thread-modular reasoning (Sect. 6.2) yields a control- and local-stateless thread-modular summary that over-approximates the reachable states of the original program. In addition, it retains the number of threads in a specific control location in the instrumented counter variables. As we motivated in Sect. 1, this is essential for constructing counting proofs of parameterized software. Observe the following property of the combination of CCA and TMS:

Proposition 3 *Let P be a program over global and local variables* $\mathbf{g}$ *and* $\mathbf{l}$. *Note that the composition of* TMS and CCA, $\mathsf{TMS}(\mathsf{CCA}(P,k))$, *ranges over additional counters* $\mathbf{c}$, i.e., *over global variables* $\mathbf{g} \cup \mathbf{c}$. *Up to these instrumentation variables*, $\mathsf{TMS}(\mathsf{CCA}(P,k))$ *over-approximates the transition relation of P's k-times interleaving P^k when projected onto global variables for all* $k > 0$, *i.e.*,

$$\mathrm{GTR}\,(P^k) \;\subseteq\; \mathrm{GTR}\,(\mathsf{TMS}(\mathsf{CCA}(P,k)))\!\downharpoonright_{\mathbf{g}} \quad \text{for all } k > 0. \tag{23}$$

Proof sketch The claim is a direct consequence of Propositions 1 and 2 (instantiating P in Proposition 2 with $\mathsf{CCA}(P,k)$).

Corollary 3 *Up to instrumentation variables*, $\mathsf{TMS}(\mathsf{CCA}(P,k))$ *over-approximates the reachable global states of P's k-times interleaving P^k for all $k > 0$, i.e.*,

$$\mathrm{GReach}\,(P^k) \;\subseteq\; \mathrm{GReach}\,(\mathsf{TMS}(\mathsf{CCA}(P,k)))\!\downharpoonright_{\mathbf{g}} \quad \text{for all } k > 0. \tag{24}$$

Application to safety proving. Recall from Definition 2 that safety of a parameterized program $P(n)$ is defined with respect to a generator set of error states Err_m. For deciding if a program state belongs to Err_m, the control locations and valuations of local variables of the $n - m$ other symmetric threads are irrelevant. We thus use the following generalization of thread-modular reasoning: We pick a finite set of m reference threads (recall that the parallel composition of finitely many threads is again a sequential program) and apply a combination of control counter abstraction and thread-modular summary generation to abstract all $n - m$ other threads.

Definition 6 (*Thread-modular counter abstraction* TMCA) Let $T[N]$ be a program template and let $P(n)$ be the induced parameterized program. Let Err_m be a generator set of error states. We define the *thread-modular counter abstraction* $\mathsf{TMCA}(T[N], n, m)$ as the program

$$\begin{aligned} \mathsf{TMCA}(T[N], n, m) \overset{\mathsf{def}}{=} & \text{ let P = T[N/n] in} \\ & P_1 \parallel \cdots \parallel P_m \parallel \mathsf{TMS}(\mathsf{CCA}(P, n - m)). \end{aligned} \tag{25}$$

Proposition 4 *Let $T[N]$ be a program template, let $P(n)$ be its induced parameterized program, let ℓ_0 and Init denote the initial states of $P(n)$. Let Err_m be a generator set of error states. We define* Reach_m *to be the set of reachable states* $\text{Reach}(P(n))$ of $P(n)$ *where its local state is projected onto the first m components, i.e., let*

$$\begin{aligned} \text{Reach}_m = \{ & ((\ell_1, \dots, \ell_m), \nu(\mathbf{g}, \mathbf{l}_1, \dots, \mathbf{l}_m)) \mid \text{s.t.} \\ & ((\ell_1, \dots, \ell_n), \nu(\mathbf{g}, \mathbf{l}_1, \dots, \mathbf{l}_n)) \in \text{Reach}(P(n), \ell_0, Init)\}. \end{aligned} \tag{26}$$

Then, $\mathsf{TMCA}(T[N], n, m)$ (*with the auxiliary instrumentation variables projected away*) *over-approximates* Reach_m, *i.e., we have that*

$$\text{Reach}_m \subseteq \text{Reach}(\mathsf{TMCA}(T[N], n, m)) \restriction_{(\mathbf{g} \cup \mathbf{l}_1 \cup \cdots \cup \mathbf{l}_m)}. \tag{27}$$

Proof sketch Let $P = T[N/n]$. We note that $P^n = P_1 \parallel \cdots \parallel P_m \parallel P^{(n-m)}$. The claim then follows from Proposition 3 because every run of P^n gives rise to a run of $P_1 \parallel \cdots \parallel P_m \parallel \mathsf{TMS}(\mathsf{CCA}(P, n - m))$ that agrees on the values of $\mathbf{g} \cup \mathbf{l}_1 \cup \cdots \cup \mathbf{l}_m$.

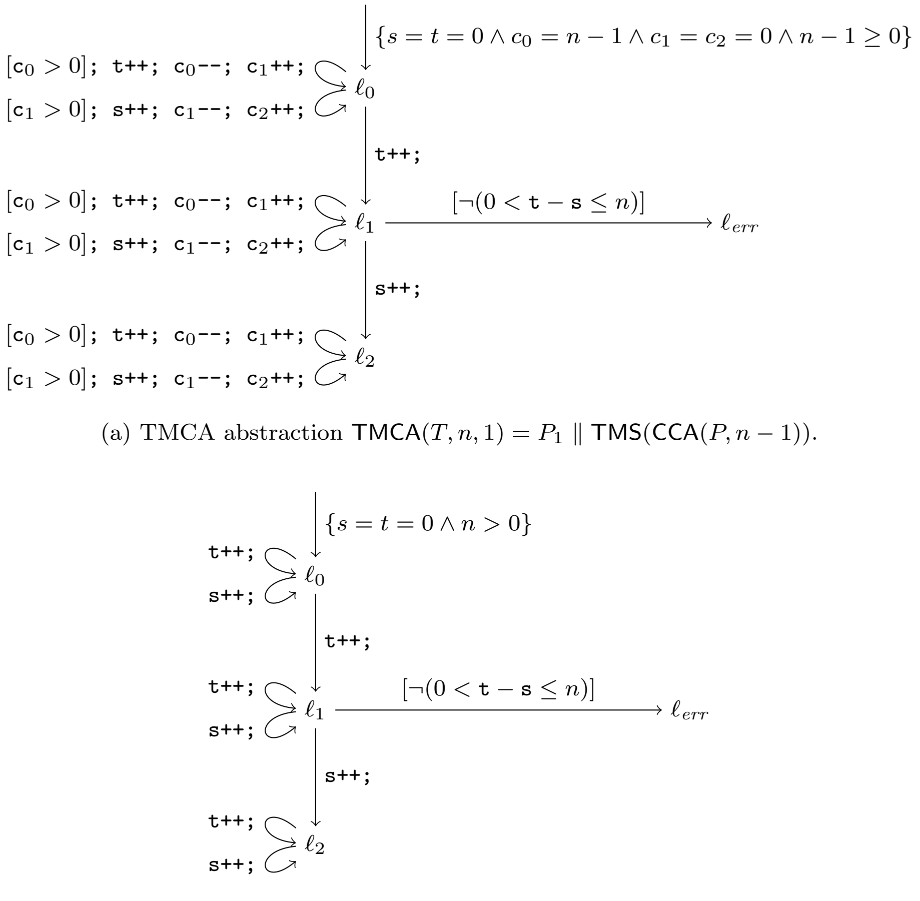
Proposition 5 *By symmetry of* $\mathsf{Err}(n)$, Reach_m *contains an error state if and only if an error state defined by* Err_m *is reachable by* $P(n)$.

Finally, this allows us to state the soundness of TMCA for safety:

Theorem 1 (Soundness for safety) *Let $T[N]$ be a program template, let $P(n)$ be its induced parameterized program, and let* Err_m *be a generator set of error states. If* $\mathsf{TMCA}(T, n, m)$ *is safe with respect to* Err_m, *then so is $P(n)$ for all* $n > 0$.

Proof Direct consequence of Propositions 4 and 5. □

The result in Theorem 1 gives us a technique for verifying the safety of a given parameterized program: Our method computes its TMCA abstraction and checks the safety of the resulting abstract program. If this program is safe, then so is the parameterized program for all system sizes $n > 0$.


$\{s = t = 0 \wedge c_0 = n-1 \wedge c_1 = c_2 = 0 \wedge n-1 \geq 0\}$
$[c_0 > 0]$; t++; c_0--; c_1++;
$[c_1 > 0]$; s++; c_1--; c_2++;
ℓ_0
t++;
ℓ_1
$[\neg(0 < \mathtt{t} - \mathtt{s} \leq n)]$
ℓ_{err}
s++;
ℓ_2
$\{s = t = 0 \wedge n > 0\}$
t++;
s++;

(a) TMCA abstraction $\mathsf{TMCA}(T, n, 1) = P_1 \parallel \mathsf{TMS}(\mathsf{CCA}(P, n-1))$.

(b) Standard thread-modular abstraction $\mathsf{TM}(T, n, 1) = P_1 \parallel \mathsf{TMS}(P)$.

Fig. 5 Running example illustrating our thread-modular abstraction TMCA next to standard thread-modular abstraction TM

6.4 Thread-modular counter abstraction (TMCA) vs. standard thread-modular reasoning (TM)

A similar result holds if we adapt standard thread-modular reasoning (TM) (e.g., [29, 40, 47]) to parameterized systems.

Definition 7 (*Standard thread-modular abstraction* TM) Let $T[N]$ be a program template and let $P(n)$ be the induced parameterized program. Let Err_m be a generator set of error states. Let program $P = T[N/n]$ and let the auxiliary program TM' be

$$\mathsf{TM}'(T[N], n, m) = (Loc, \delta, \ell_0, Init) \overset{\text{def}}{=} P_1 \parallel \cdots \parallel P_m \parallel \mathsf{TMS}(P). \tag{28}$$

The *standard thread-modular abstraction* $\mathsf{TM}(T[N], n, m)$ is the program TM' with an augmented initial state constraint on the system size n, i.e.,

$$\mathsf{TM}(T[N], n, m) \overset{\text{def}}{=} (Loc, \delta, \ell_0, Init') \quad \text{where } Init' \overset{\text{def}}{=} Init \wedge n > 0. \tag{29}$$

Similar to TMCA, standard thread-modular reasoning TM over-approximates the states reachable by the parameterized program and is sound for safety:

Proposition 6 *Let* Reach_m *be defined as in Proposition 27.* $\mathsf{TM}(T[N], n, m)$ *over-approximates* Reach_m*, i.e., we have that* $\text{Reach}_m \subseteq \text{Reach}\,(\mathsf{TM}(T[N], n, m))$.

Proof sketch Let $P = T[N/n]$. We note that $P^n = P_1 \parallel \cdots \parallel P_m \parallel P^{(n-m)}$. The claim then follows from Proposition 2 because every run of P^n gives rise to a run of $P_1 \parallel \cdots \parallel P_m \parallel \mathsf{TMS}(P))$ that agrees on the values of $\mathbf{g} \cup \mathbf{l}_1 \cup \cdots \cup \mathbf{l}_m$.

Theorem 2 (Soundness of standard thread-modular reasoning) *Let $T[N]$ be a program template, let $P(n)$ be its induced parameterized program, and let* Err_m *be a generator set of error states. If* $\mathsf{TM}(T, n, m)$ *is safe with respect to* Err_m*, then so is $P(n)$ for all $n > 0$.*

Proof Direct consequence of Propositions 6 and 5. □

While both standard thread-modular reasoning TM and our TMCA abstraction TMCA are sound, TMCA is a stronger abstraction than TM. Below, we give an example where our TMCA abstraction is capable of proving safety and standard TM is not.

Running example (Safety) Figure 5a shows the TMCA abstraction (Definition 25) of our running example (continued from Fig. 4c) next to the standard thread-modular abstraction TM (Definition 29) in Fig. 5b. Note that the only difference between the two abstract programs is TMCA's stronger environment transition relation, which is introduced by the auxiliary counter instrumentation CCA (Definition 17). Yet, in the TMCA abstraction the error location ℓ_{err} is unreachable, while the TM abstraction is unsafe.

This is witnessed by the following feasible error path on TM (Fig. 5b) that invalidates the assertion's lower bound (starting in an initial state where $s = t = 0 \wedge n > 0$):

$$\ell_0 \xrightarrow{\texttt{t++}} \ell_1 \xrightarrow{\texttt{s++}} \ell_1 \xrightarrow{[\neg(0<\texttt{t}-\texttt{s}\leq n)]} \ell_{err}. \tag{30}$$

Similarly, the assertion's upper bound is invalidated, e.g., for $n = 2$ by the following feasible error path on TM (Fig. 5b):

$$\ell_0 \xrightarrow{\texttt{t++}} \ell_1 \xrightarrow{\texttt{t++}} \ell_1 \xrightarrow{\texttt{t++}} \ell_1 \xrightarrow{[\neg(0<\texttt{t}-\texttt{s}\leq n)]} \ell_{err}. \tag{31}$$

Both error paths are infeasible in the TMCA abstraction (Fig. 5a): In the first case, the transition corresponding to $\ell_1 \xrightarrow{\texttt{s++}} \ell_1$ is infeasible because $\mathsf{c}_1 = 0$, i.e., no environment thread has yet moved from ℓ_0 to ℓ_1. In the second case, the number of times the transition corresponding to $\ell_1 \xrightarrow{\texttt{t++}} \ell_1$ can be taken is bounded by the initial value of c_0, i.e. $n - 1 = 1$.

Theorem 3 (Expressiveness) *Let* Reach_m *be defined as in Proposition 27. For all program templates $T[N]$,* TMCA *is a stronger abstraction than* TM*, i.e., we have that*

$$\begin{aligned} \text{Reach}_m &\subseteq \text{Reach}\,(\mathsf{TMCA}(T[N], n, m))\downharpoonright_{(\mathbf{g} \cup \mathbf{l}_1 \cup \cdots \cup \mathbf{l}_m)} \\ &\subseteq \text{Reach}\,(\mathsf{TM}(T[N], n, m)). \end{aligned}$$

Proof sketch We observe that $\mathsf{TMS}(\mathsf{CCA}(P, n-m)) \subseteq \mathsf{TMS}(P)$ because every run of $\mathsf{CCA}(P, n-m)$ gives rise to a run of P when projecting away the instrumented variables. Hence, every run of $\mathsf{TMCA}(T[N], n, m)$ gives rise to a run of $\mathsf{TM}(T[N], n, m)$, and the claim follows.

Discussion. It is well-known that standard thread-modular reasoning effectively abstracts the following three aspects of multi-threaded programs [73]:

1. the order in which program transitions happen (*transition order*),
2. how many times each transition is performed (*transition multiplicity*), and
3. which thread takes which transition (*thread identities*).

It is also well-known that adding auxiliary state to a thread-modular abstraction increases expressiveness [54, 55]. TMCA's auxiliary counter instrumentation effectively introduces *some* information about both *transition order* and *transition multiplicity* on top of standard thread-modular reasoning. Although TMCA is equally oblivious to thread identities, in our experience this is not an issue due to the symmetry of parameterized programs.

6.5 Conclusion

In this section we introduced our thread-modular counter abstraction TMCA. We showed that safety of the TMCA abstraction implies safety of the parameterized program for all system sizes, and compared TMCA to standard thread-modular reasoning. In the next section, we discuss how to automatically discharge a safety proving obligation on our TMCA abstraction.

7 Tackling infinity dimension II: predicate abstraction (PA)

The parameterized program $P(n)$ induced by a program template $T[N]$ refers to an *infinite family of programs*. In contrast, consider its *thread-modular counter abstraction* $\mathsf{TMCA}(T, n, m)$: if its parameter n remains symbolic, we obtain an abstraction of the parameterized program in the form of a sequential program with finitely many control locations and local variables, while over-approximating the infinite family of programs induced by $P(n)$. Standard software verification methods could be applied to prove safety, thus tackling Infinity Dimension (II) from Sect. 1.3: the *infinite data domain*.

Divergence of predicate abstraction-based verifiers. However, our experiments in Sect. 8 show that standard methods often fail on our models: We encode the TMCA abstraction of our benchmarks as a set of constrained Horn clauses [36] (CHCs). Both state-of-the-art solvers Eldarica [43] and Z3 [20] diverge on many of our examples (Table 1, columns 1c and 1d; cf. Sect. 8 for details). We speculate that this is due to the uncommon structure of our TMCA models; each control location of our abstraction has self-loops of the thread-modular summary attached (compare, e.g., Fig. 4c). In this section, we discuss how to guide a predicate abstraction-based solver to converge on TMCA models.

$$\begin{aligned}
&s = 0 \wedge t = 0 \wedge c_0 = n - 1 \wedge c_1 = 0 \wedge c_2 = 0 \wedge n > 0 \wedge && \text{(initial state)}\\
&s' = s \wedge t' = t + 1 \wedge c_0' = c_0 \wedge c_1' = c_1 \wedge c_2' = c_2 \wedge && (\ell_0 \to \ell_1\text{: } \mathtt{t{+}{+}})\\
c_0' > 0 \wedge {}&s'' = s' \wedge t'' = t' + 1 \wedge c_0'' = c_0' - 1 \wedge c_1'' = c_1' + 1 \wedge c_2'' = c_2' \wedge && (\ell_1 \to \ell_1\text{: } \mathsf{IncT})\\
c_1'' > 0 \wedge {}&s''' = s'' + 1 \wedge t''' = t'' \wedge c_0''' = c_0'' \wedge c_1''' = c_1'' - 1 \wedge c_2''' = c_2'' + 1 \wedge && (\ell_1 \to \ell_1\text{: } \mathsf{IncS})\\
&0 < t''' - s''' && \text{(assertion)}
\end{aligned}$$

(a) Concrete interpolation query.

$$\begin{aligned}
&s = 0 \wedge t = 0 \wedge c_0 = n - 1 \wedge c_1 = 0 \wedge c_2 = 0 \wedge n > 0 \wedge && \text{(initial state)}\\
&s' = s \wedge t' = t + 1 \wedge c_0' = c_0 \wedge c_1' = c_1 \wedge c_2' = c_2 \wedge && (\ell_0 \to \ell_1\text{: } \mathtt{t{+}{+}})\\
c_0' > 0 \wedge {}&s^A = s' \wedge t^A = t' + 1 \wedge c_0^A = c_0' - 1 \wedge c_1^A = c_1' + 1 \wedge c_2^A = c_2' \wedge \boxed{(s^A = \dot{s} \wedge t^A - c_1^A = \dot{t} - \dot{c}_1)} \wedge && (\ell_1 \to \ell_1\text{: } \mathsf{IncT})\\
c_1^B > 0 \wedge {}&s''' = s^B + 1 \wedge t''' = t^B \wedge c_0''' = c_0^B \wedge c_1''' = c_1^B - 1 \wedge c_2''' = c_2^B + 1 \wedge \boxed{(s^B = \dot{s} \wedge t^B - c_1^B = \dot{t} - \dot{c}_1)} \wedge && (\ell_1 \to \ell_1\text{: } \mathsf{IncS})\\
&0 < t''' - s''' && \text{(assertion)}
\end{aligned}$$

(b) Abstract interpolation query.

Fig. 6 Interpolation queries for our running example from Fig. 4c

7.1 Predicate selection for TMCA models

A standard method for building predicate abstractions is to iteratively use an interpolating theorem prover to find new predicates that rule out spurious counter-examples [52]: We encode the error path in a logical formula in the usual way and split it into partitions $A \wedge B$. If the formula is unsatisfiable, the solver returns an *interpolant* I over the common symbols of A and B such that $A \rightarrow I$ and $I \rightarrow \neg B$. Intuitively, the interpolant I gives a reason why the path $A \wedge B$ is infeasible, and can thus be used as a predicate to refine the abstraction.

Choosing good predicates. The key to converging predicate abstraction CEGAR loops is to chose the "right" interpolants. *Loop counters* are variables that are incremented or decremented on a loop path of the program. Conventional wisdom holds that referring to such loop counters, which frequently appear on infeasible error paths, is best avoided in abstract models: tracking their values leads to loop unrolling and divergence of the CEGAR loop [13, 68]. This poses a challenge for thread-modular summaries, as we demonstrate on our running example:

Running example (Safety) Recall the TMCA abstraction of our example in Fig. 4c: Due to product construction with the thread-modular summary

$$\mathsf{TMS}(\mathsf{CCA}(P, n-1)), \tag{32}$$

all variables are loop counters: the self-loops IncS and IncT at each program location increment or decrement $\mathtt{c}_0$, $\mathtt{c}_1$, $\mathtt{c}_2$, $\mathtt{s}$, and $\mathtt{t}$. Tracking the value of either one leads to useless loop unrollings.

Even more elaborate predicates, e.g., tracking the difference expression $t - s$ in the assertion do not lead to convergence: Assume that we already applied predicate abstraction and the model checker returned the following spurious counter-example[5] (starting in an initial state where $s = t = 0$):

$$\mathtt{t}{+}{+};\ \mathsf{IncT};\ \mathsf{IncS};\ [0 \geq \mathtt{t} - \mathtt{s}]; \tag{33}$$

[5] One can reproduce the behavior of this running example in the model checker Eldarica (v2.0.2) [43] and the interpolating theorem prover Princess (v2020-03-12) [67].

Table 1 Benchmark results: time to prove safety

	(1) TMCA abstraction (Section 6)				(2) Pacman	(3) Eldarica
	(a) our heuristic	(b) Eldarica	(c) Eldarica	(d) Z3		
	(Section 7)	`-abstract:relIneqs`	`-abstract:off`			
(A) Global variable increment / decrement						
pp	1.5s	1.5s	1.4s	**0.1s**		⏲
mm	1.5s	1.7s	1.4s	**0.1s**		⏲
ppmm	2.5s	**2.3s**	⏲	⏲		⏲
mmpp	2.6s	**2.3s**	⏲	⏲		⏲
ppmmpp	**95.5s**	179.1s	⏲	⏲		⏲
(B) SV-COMP						
fkp2014 [25]	**2.0s**	⏲	⏲	⏲		⏲
fkp2014 extd. (Fig. 1c)	**2.0s**	⏲	⏲	⏲		⏲
qw2004 [25]	**2.7s**	5.5s	⏲	⏲		⏲
(C) Synchronization barriers from [31]						
locals [31]	**124.6s**	⏲	⏲	⏲	16s	⏲
shareds [31]	23.8s	**10.9s**	⏲	⏲	160s	⏲
readflag [31]	**25.5s**	⏲	⏲	⏲	34s	⏲
semaphore [31]	**36.4s**	⏲	⏲	⏲	68s	⏲
cyclic [31]	7.3s	**4.5s**	4.9s	⏲	30s	⏲
maximum [31]	no thread-modular proof				489s	⏲
parent-child [31]	dynamic thread creation				76s	dyn.thr.c.
as-many [31]	dynamic thread creation				68s	dyn.thr.c.

⏲indicates a timeout after 15 min, the fastest tool for our TMCA encoding is highlighted in bold

The formula representing this error path is shown in Fig. 6a. If we partition the formula between IncT and IncS, an interpolating theorem prover is likely to find the new predicate $2 \leq t - s$. This rules out the spurious counter-example above, but leads to another, longer one:

$$\mathtt{t}++;\ \mathsf{IncT};\ \mathsf{IncT};\ \mathsf{IncS};\ \mathsf{IncS};\ [0 \geq \mathtt{t} - \mathtt{s}]; \tag{34}$$

This again can be ruled out by the additional predicate $3 \leq t - s$ but only leads to further unrollings of IncS and IncT and to further invariants of this shape; the CEGAR loop diverges.

Finding better interpolants. Instead, we want to find an invariant that relates the location counters $\mathtt{c}_0$, $\mathtt{c}_1$, $\mathtt{c}_2$ to the values of the global variables $\mathtt{s}$ and $\mathtt{t}$. Thus, traditional predicate selection heuristics do not apply, and we need a new selection procedure suitable for TMCA models (and similarly structured programs). The next section explains how to achieve this.

7.2 An interpolation abstraction heuristic for TMCA models

As we argued above, interpolating predicate abstraction is *always* driven by heuristics to prevent divergence. We now present a heuristic that we find useful for the considered problem domain and later show that it outperforms several existing ones (for experimental evidence, cf. Sect. 8).

Interpolation abstraction [51] is a state-of-the-art method to implement predicate selection. The technique uses a set of *template terms* T to abstract the interpolation query and thus guide the theorem prover in its search for an interpolant. Template terms represent relations between common variables $\mathbf{v}$ of the partitions A and B of an interpolation problem. An abstraction of the interpolation query $A \wedge B$ is obtained by renaming the common symbols $\mathbf{v}$ in A and B to $\mathbf{v}^A$ and $\mathbf{v}^B$, respectively, and subsequently constraining the resulting formulas with the equalities $\dot{\mathbf{v}} = \mathbf{v}^A$ and $\dot{\mathbf{v}} = \mathbf{v}^B$ and instantiations of the template terms T over $\dot{\mathbf{v}}$. Following [51], we use $T(A) \wedge T(B)$ to denote the resulting abstract interpolation query and call the abstract query *feasible* if $T(A) \wedge T(B)$ is still unsatisfiable. By construction, any interpolant for a feasible interpolation abstraction $T(A) \wedge T(B)$ corresponds to an interpolant for the original query $A \wedge B$ (but not vice versa).

Eldarica with its default interpolation abstraction heuristic (Table 1, column 1b) already fares better than without (column 1c) but still diverges on some benchmarks. We introduce a dedicated heuristic for TMCA models to remedy this shortcoming by encoding our domain knowledge in the template terms T. We briefly introduce interpolation abstraction on our running example and refer the interested reader to the canonical description [51] for further reading.

***Running example* (*Safety*)** As explained in Sect. 1, the valuations of `s` and `t` correspond to the number of threads in specific control locations, and thus to sums over the instrumented location counters. In particular, at ℓ_1 we have that

$$t = c_1 + c_2 + 1 \quad \text{and} \quad s = c_2 \quad \text{and thus} \tag{35}$$

$$t - s = (c_1 + c_2 + 1) - (c_2) = c_1 + 1. \tag{36}$$

Assume that we choose template terms $\{t - c_1, s\}$. The abstracted query is shown in Fig. 6b: Common symbols at the interpolation point have been renamed and limited knowledge about them is reintroduced via equalities over the template terms in the shaded subformulae: in particular, the concrete values of t'' and c_1'' are lost, and only relational knowledge about their difference is reintroduced. Thus, $2 \leq \dot{t} - \dot{s}$ is no longer an interpolant. Instead, our interpolation procedure finds the new predicate $c_1 < t - s$, which is inductive at ℓ_1 and rules out further unrollings of the thread-modular summary. Note that this predicate $c_1 < t - s$ is implied by the invariant in Eq. (36) and, together with $0 \leq c_1$, implies the assertion $0 < t - s$.

It remains to define how our method computes the set of template terms for interpolation abstraction.

Definition 8 (*Interpolation abstraction template terms*) Let $T[N]$ be a program template over global and local variables $\mathbf{g}$ and $\mathbf{l}$, let $P = T[N/n]$ be the program obtained by replacing N in T with n, and let $P(n)$ be the induced parameterized program. We start by computing a set of template terms for the thread-modular abstraction $\mathsf{TMS}(\mathsf{CCA}(P, n - m))$. For each variable x, we compute a stride set

$$S(x) = \{\alpha \mid x \text{ is incremented by } \alpha \text{ on some transition of } \mathsf{TMS}(\mathsf{CCA}(P, n - m))\}. \tag{37}$$

We then define difference terms

$$T_{\mathsf{TMS}} = \{\alpha x - \beta c \mid \text{x is a global program variable}, \\ c \text{ is a location counter introduced by } \mathsf{CCA}, \\ \alpha \in S(c) \text{ and } \beta \in S(x)\}. \tag{38}$$

We define the set of interpolation abstraction template terms *Templ* as the union of the following:

1. all global variables $\mathbf{g}$,
2. the parameter n,
3. the set of difference terms T_{TMS}.

Searching the abstraction space. We replace the template term heuristics of [51] with our set *Templ*. The powerset lattice $\langle 2^{Templ}, \subseteq\rangle$ over the template terms then induces a search space of interpolation abstractions, which is systematically explored by the abstraction algorithm from [51]. Intuitively, smaller subsets correspond to logically stronger abstractions. Larger subsets of *Templ* may cause the abstraction to become too weak, i.e., cause the interpolation query to become satisfiable. Thus, the search algorithm of [51] first explores the subset lattice to find the maximal elements (viz. the largest subsets of template terms *Templ*) for which the interpolation query is still unsatisfiable. Let *Cand* refer to this candidate set of strongest possible abstractions. The algorithm then assigns a cost to each candidate abstraction $C \in Cand$. [51] explores a number of cost functions; we choose a simple one that counts the number of template terms in c, i.e., it assigns uniform cost to each element in C. Given this cost-weighted set of maximal unsatisfiable candidate abstractions, the algorithm of [51] picks the ones with minimal cost (in our case, the smallest subsets in *Cand*) to abstract the interpolation query and compute interpolants (and thus refinement predicates).

Our template terms in abstraction search. Intuitively, our choice of template terms *Templ* leads the search to explore relational abstractions, such as $t - c_1$, early by assigning them the same cost as selecting a single global variable $g \in \mathbf{g}$. Moreover, it still allows us to track the value of global variables and to introduce the parameter n if necessary. In cases where there is no relationship between the global variables and location counters as captured by T_{TMS}, our templates may still be useful by ruling out interpolants that track concrete variable values and would lead to loop unwinding. Finally, even though our template terms are linear relations, interpolation abstraction is semantic in nature and does not restrict the prover to only find such interpolants [51].

8 Safety experiments

We implement our TMCA abstraction and predicate discovery engine inside the Eldarica safety verifier [43, 51] as Eldarica with TMCA[23]. Our extension takes as input a program template $T[N]$ and the error states Err_m in a C-like language and outputs the abstracted program $\mathsf{TMCA}(T, n, m)$ as a set of constrained Horn clauses (CHCs) [36] in the standard SMT-LIB format. To this output we apply different CHC solvers, including Eldarica's own Horn solver with its default and our customized predicate selection heuristic.

8.1 Benchmarks

Our safety benchmarks and results are shown in Table 1. The first group of benchmarks (A) [11] consists of program templates that sequentially increment and decrement a global variable. At each program location we assert the tightest possible lower and upper bounds; given that the number of increments and decrements depends on the number of concurrent threads n, these assertions are parameterized by the number of concurrent threads.

The second group of benchmarks (B) is a set of programs using unbounded thread creation taken from the software verification competition SV-COMP [12]. In its latest three editions (2018–2020), no sound verification tool proved these benchmarks safe. In addition, `fkp2014` and the bluetooth driver `qw2004` are the introductory and running example of [25].

The third group of benchmarks (C) from [30, 57] includes non-monotonic synchronization barriers (cf. Sect. 2).

8.2 Comparisons

The columns of Table 1 compare our two main contributions for safety, i.e., approaches to

1. **Parameterized verification** We compare *our TMCA abstraction* (column 1) combined with different backend solvers (sub-columns 1a–1d) to *other parameterized program verifiers* (columns 2 and 3).
2. **Predicate selection.** On our TMCA abstraction (column 1), we compare *our predicate selection heuristic* (sub-column 1a) to *other predicate selection heuristics* (sub-columns 1b–1d).

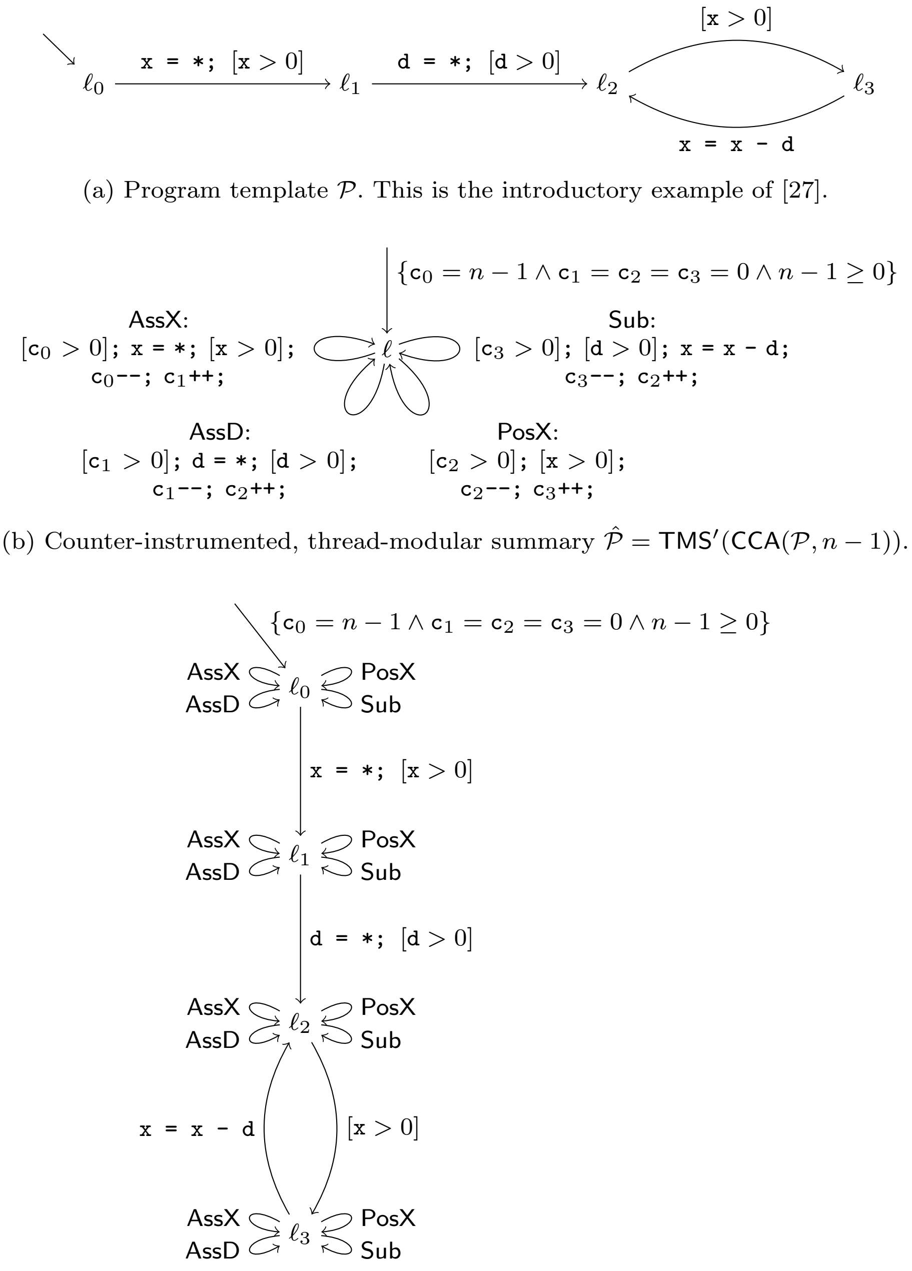
Parameterized verification: TMCA vs. others In particular, we first compare TMCA abstraction (column 1) to other parameterized program verifiers: Pacman [30] (column 2) and Eldarica's unbounded thread encoding[6] [44] (column 3).

Predicate selection: our heuristic vs. others Second, we compare different CHC solvers on our TMCA-abstracted models in the subcolumns of column 1: our predicate selection heuristic from Sect. 7 (column 1a), Eldarica's default heuristic [51] (`-abstract:relIneqs`, column 1b), Eldarica without interpolation abstraction (`-abstract:`off, column 1c) and the CHC solver in Z3 [20] (column 1d).

8.3 Results

Parameterized verification The last two benchmarks, `parent-child` and `as-many`, use dynamic thread creation which is currently not supported by Eldarica. Eldarica times out on the remaining ones. Unfortunately, we were unable to replicate the results for Pacman from [30]. We are thus limited to citing previous results from [30] (recall from Sect. 2

[6] This encoding is usually unaware of the parameter n. We therefore slightly modify our benchmarks such that the encoding's implicitly introduced local thread id variable is bounded by n.


ℓ_0
x = *; $[\mathtt{x} > 0]$
ℓ_1
d = *; $[\mathtt{d} > 0]$
ℓ_2
$[\mathtt{x} > 0]$
ℓ_3
x = x - d
$\{\mathtt{c}_0 = n - 1 \wedge \mathtt{c}_1 = \mathtt{c}_2 = \mathtt{c}_3 = 0 \wedge n - 1 \geq 0\}$
ℓ
AssX:
$[\mathtt{c}_0 > 0]$; x = *; $[\mathtt{x} > 0]$;
$\mathtt{c}_0$--; $\mathtt{c}_1$++;
Sub:
$[\mathtt{c}_3 > 0]$; $[\mathtt{d} > 0]$; x = x - d;
$\mathtt{c}_3$--; $\mathtt{c}_2$++;
AssD:
$[\mathtt{c}_1 > 0]$; d = *; $[\mathtt{d} > 0]$;
$\mathtt{c}_1$--; $\mathtt{c}_2$++;
PosX:
$[\mathtt{c}_2 > 0]$; $[\mathtt{x} > 0]$;
$\mathtt{c}_2$--; $\mathtt{c}_3$++;
$\{\mathtt{c}_0 = n - 1 \wedge \mathtt{c}_1 = \mathtt{c}_2 = \mathtt{c}_3 = 0 \wedge n - 1 \geq 0\}$
AssX
AssD
ℓ_0
PosX
Sub
x = *; $[\mathtt{x} > 0]$
AssX
AssD
ℓ_1
PosX
Sub
d = *; $[\mathtt{d} > 0]$
AssX
AssD
ℓ_2
PosX
Sub
x = x - d
$[\mathtt{x} > 0]$
AssX
AssD
ℓ_3
PosX
Sub

(a) Program template $\mathcal{P}$. This is the introductory example of [27].

(b) Counter-instrumented, thread-modular summary $\hat{\mathcal{P}} = \mathsf{TMS}'(\mathsf{CCA}(\mathcal{P}, n-1))$.

(c) TMCA abstraction $\mathsf{TMCA}(\mathcal{P}, n, 1) \ = \ \mathcal{P}_1 \parallel \mathsf{TMS}'(\mathsf{CCA}(\mathcal{P}, n-1))$.

Fig. 7 Termination running example

that our main objective is to replace their dedicated abstraction techniques with a cleaner framework of well-established ones).

Parameterized verification nOf the remaining benchmarks, only `maximum` does not have a thread-modular proof and thus cannot be proved safe by our method. On the remaining benchmarks, TMCA (Sect. 6) combined with our predicate selection heuristic (Sect. 7) is the only toolchain to solve all tasks. It does so well below the timeout limit of 15 min. Meanwhile, Eldarica with default predicate selection heuristics encounters 5 timeouts, Eldarica without interpolation abstraction 10, and Z3 in its standard configuration even 11 timeouts. This shows how indispensable an appropriate predicate discovery algorithm is for our thread-modular abstractions.

Conclusion In summary, a combination of both our contributions for safety (TMCA abstraction and our predicate selection heuristic) is necessary to tackle all benchmarks; no other toolchain comes close in results.

This concludes our discussion on safety verification; in the next section, we discuss a generalization of our methods to termination proving.

9 Case study: generalization to termination

So far, our technical exposition has focused on *safety properties*. In contrast, *termination* is a foundational liveness property: like reachability for safety, proving liveness can be reduced to proving termination [15, 71]. Unlike termination proving for sequential programs (e.g., [14, 17, 19, 24, 34]) and multi-threaded programs with a fixed number of threads (e.g., [3, 4, 36, 66]), which have seen an ample amount of research in the past, liveness verification of parameterized programs has only recently become a focus of research (cf. Sect. 2). In this section, we present a generalization of our TMCA-based analysis to termination.

9.1 Introduction

We start our discussion with a case study, developing our technique on our introductory example for termination analysis from Fig. 2. For easier reference, we also show this program template in Fig. 7a.

Running example (Termination) Recall program $\mathcal{P}$ (Fig. 7a) from our motivating example in Sect. 4: First, `x` and then `d` are assigned positive integer values. The subsequent loop subtracts `d` from `x` until `x` becomes non-positive. We are interested in showing termination of the parameterized program $\mathcal{P}(n) = \mathcal{P}_1 \parallel \cdots \parallel \mathcal{P}_n$ composed of n threads executing $\mathcal{P}$ for all $n > 0$.

Figure 7b shows the counter-instrumented, thread-modular summary

$$\hat{\mathcal{P}} = \mathsf{TMS}'(\mathsf{CCA}(\mathcal{P}, n-1)) \tag{39}$$

of $\mathcal{P}$. Note that here we adopt TMS', a slightly stronger version of TMS that restricts the summary transitions to reachable states. This allows us to include the additional assumption that $\mathtt{d}$ is positive $[\mathtt{d} > 0]$ in transition Sub. This is in accordance with our definition of thread-modular summaries (Definition 4). Methods for computing such summaries have been described throughout the literature, for example in [69, 70].

Finally, consider the thread-modular counter abstraction $\mathsf{TMCA}(\mathcal{P}, n, 1) = \mathcal{P}_1 \parallel \hat{\mathcal{P}}$ (Fig. 7c). We show that this abstraction terminates, and thus also the parameterized program $\mathcal{P}(n)$ terminates for all $n > 0$. We give an intuitive termination proof for the TMCA abstraction in Fig. 7c:

Note that the concrete thread $\mathcal{P}_1$ can only diverge if it gets stuck in the loop $\ell_2 \to \ell_3 \to \ell_2 \to \ldots$. Without interference from the TMCA-abstracted environment $\hat{\mathcal{P}}$, $\mathcal{P}_1$ terminates because $\mathtt{d}$ is strictly positive at ℓ_2 and ℓ_3, and thus $\mathtt{x}$ decreases in each iteration towards the bound $[\mathtt{x} > 0]$.

Let us now consider interference from the thread-modular summary $\hat{\mathcal{P}}$: Transitions $\mathsf{AssX}, \mathsf{AssD}, \mathsf{PosX}, \mathsf{Sub}$ do not change the fact that $\mathtt{d}$ is strictly positive once the concrete thread $\mathcal{P}_1$ reaches ℓ_2. The remaining environment interference that could cause the program to diverge is transition AssX that resets the value of $\mathtt{x}$. At this point, TMCA's counter instrumentation comes into play. While AssX resets the value of $\mathtt{x}$ and may thus prevent its value from converging towards 0, the number of times transition AssX can be executed is unbounded but finite: due to its guard $[\mathtt{c}_1 > 0]$, this transition may not execute more than $n - 1$ times. This symbolic expression is itself unbounded from above, but only takes finite, non-negative values. After at most $n - 1$ executions of AssX, the value of $\mathtt{x}$ cannot be reset anymore, and progresses towards zero in the loop $\ell_2 \to \ell_3 \to \ell_2 \to \ldots$ as above (and potentially also by interleaved executions of PosX and Sub by the abstracted environment $\hat{\mathcal{P}}$).

Finally, we observe that also the abstracted environment terminates: Transitions AssX and AssD can only be executed finitely often based on the arrangement of auxiliary counters $\mathtt{c}_0$ and $\mathtt{c}_1$. Executions of PosX and Sub may interleave and alternate, but also make the value of $\mathtt{x}$ progress towards zero, and thus cannot be executed indefinitely.

Termination of the abstraction $\mathsf{TMCA}(\mathcal{P}, n, 1) = \mathcal{P}_1 \parallel \hat{\mathcal{P}}$ together with soundness of our abstraction (argued below), implies termination of the parameterized program $\mathcal{P}(n)$ for all system sizes $n > 0$. Consider the aforementioned fact that $\mathsf{TMCA}(\mathcal{P}, n, 1)$ is a sequential program. We can thus use an existing termination prover for sequential software (e.g., [14, 17, 19, 24, 34]) to compute a termination argument for our abstraction. Below, we give a more formal description of our termination analysis.

9.2 Problem statement

Given a program template $T[N]$, we consider the induced parameterized program $P(n) = P_1 \parallel \cdots \parallel P_n$. Our goal is to show that $P(n)$ terminates from all initial states and for all system sizes $n > 0$.

Definition 9 (*Termination*) Let $T[N]$ be program template, and let $P(n)$ be its induced parameterized program. Consider an instance $P(k)$ for a scalar $k \in \mathbb{N}^+$ of $P(n)$. Recall from Definition 1 that a *run* of program $P(k)$ is an interleaved sequence of states and transitions $(\ell_0, v_0) \xrightarrow{gc_0} (\ell_1, v_1) \xrightarrow{gc_1} \ldots$. We say that program $P(k)$ *terminates* if all its runs are finite. We say that the parameterized program $P(n)$ *terminates* if all its instances $P(1), P(2), \ldots$ terminate.

As for safety, this means proving that *each member* of the *infinite family* $\{P(1), P(2), \dots\}$ of instances induced by the parameterized program $P(n)$ is terminating.

9.2.1 Equivalent notions

We briefly consider the notion of *thread termination* [18], and how it relates to parameterized program termination.

Definition 10 (*Thread termination*) Let $T[N]$ be program template, and let $P(n) = P_1 \parallel \cdots \parallel P_n$ be its induced parameterized program. P_i is *thread-terminating* in the context of $P(n)$, if all runs of $P(n)$ contain only finitely-many transitions by P_i.

Note that termination of the parameterized program $P(n)$ for all $n > 0$ is equivalent to proving *thread termination* of all threads P_i of $P(n)$ for $0 < i \leq n$. If all threads P_i of $P(n)$ are thread-terminating, then $P(n)$ makes only finitely many steps and thus is itself terminating. Finally, also note that, by symmetry of $P(n)$, thread termination of P_1 implies thread termination of all P_i $(0 < i \leq n)$.

9.3 TMCA for termination

As we argued above, termination of the abstraction $\mathsf{TMCA}(P, n, 1) = P_1 \parallel \hat{P}$ together with soundness of our abstraction implies termination of the parameterized program $P(n)$ for all system sizes $n > 0$. In this section, we show that TMCA is a sound abstraction not only for safety, but also for termination.

9.3.1 Our goal

More formally, our goal is to show that if the TMCA abstraction

$$\mathsf{TMCA}(P, n, m) \tag{40}$$

terminates for some $m \geq 0$, then also the parameterized program $P(n)$ terminates. Intuitively, the counter instrumentation CCA and the thread-modular abstraction TMS must retain all behaviors of the original parameterized program, i.e., TMCA must not restrict the abstraction such that it is terminating when the original parameterized program is not.

9.3.2 Building blocks

The main building blocks of our soundness argument where already presented in Sect. 6 for safety: From Proposition 1 we have that the counter instrumentation of CCA preserves the transition relation of all k-times interleavings P^k of program P. The instrumented counters are purely auxiliar and do not alter the transition relation of the instrumented program when projected away. Following directly from the definition of a thread-modular summary (Definition 4), we have that TMS (and TMS′) over-approximates the transition relation of all k-times interleavings P^k of program P when projected onto global states (Proposition 2).

As for safety, we use this fact to obtain a single summary $\mathsf{TMS}(\mathsf{CCA}(P,k))$ that over-approximates all k-times interleavings P^k (Proposition 3).

9.3.3 Soundness for termination

This leads us to our main theorem for termination. It considers termination of the TMCA abstraction

$$\mathsf{TMCA}(P,n,m) = P_1 \parallel \cdots \parallel P_m \parallel \mathsf{TMS}(\mathsf{CCA}(P,n-m)) \tag{41}$$

for some m to prove termination of the parameterized program $P(n)$.

Theorem 4 *Let P be a program template and let $P(n)$ be its induced parameterized program. If the thread-modular abstraction $\mathsf{TMCA}(P,n,m)$ is terminating for some $m \geq 0$, then so is the parameterized program $P(n)$ for all $n > 0$.*

Proof sketch Pick an arbitrary $m \geq 0$ and assume that $\mathsf{TMCA}(P,n,m)$ is terminating. Thus, none of the first m concrete threads $P_1 \parallel \cdots \parallel P_m$ of $\mathsf{TMCA}(P,n,m)$ executes infinitely-many steps. From Proposition 3 we have that $\mathsf{TMS}(\mathsf{CCA}(P,n-m))$ over-approximates the transition relation of the remaining $n-m$ threads. Therefore, none of the first m threads $P_1 \parallel \cdots \parallel P_m$ of the parameterized program $P(n) = P_1 \parallel \cdots \parallel P_n$ execute infinitely-many steps. By symmetry of the parameterized program $P(n)$, $P(n)$ itself is terminating for all $n > 0$. In the special case of $m = 0$, $\mathsf{TMS}(\mathsf{CCA}(P,n))$ already over-approximates the transition relation of all threads, and its termination thus immediately implies termination of the parameterized program $P(n)$.

Remark The above statement is most obvious for a complete thread-modular abstraction where no concrete thread is retained ($m = 0$), or for a single concrete thread ($m = 1$) composed with the thread-modular summary. Still, the argument in Theorem 4 holds more generally for an arbitrary number of concrete threads $m \geq 0$. Preliminary work in [42] has shown that larger m result in a more expressive proof system for safety; whether this is also the case for termination and liveness is an interesting theoretical problem left for future work (cf. Sect. 10).

9.3.4 Soundness via thread termination

Inspired by the notion of thread termination discussed above (Sect. 9.2), we can state an alternate version of Theorem 4 that does not require termination of the thread-modular summary $\mathsf{TMS}(\mathsf{CCA}(P,n-m))$. Instead, we only require thread termination of the m concrete threads of $\mathsf{TMCA}(P,n,m)$:

Theorem 5 *Let P be a program template and let $P(n)$ be its induced parameterized program. If the thread-modular abstraction $\mathsf{TMCA}(P,n,m)$ is thread-terminating for $P_1,\ldots,P_m$ and some $m > 0$, then so is the parameterized program $P(n)$ for all $n > 0$.*

9.4 Automation: a case study

As a case study for automation of our approach, we encode the TMCA abstraction $\mathsf{TMCA}(\mathcal{P}, n, m)$ of our running example (Fig. 7). We prove termination of the abstraction $\mathsf{TMCA}(\mathcal{P}, n, m)$ using the termination analyzer T2 [14, 19], thus eliminating the premise of Theorem 4 and deducing termination of the parameterized program $\mathcal{P}(n)$ for all $n > 0$. We analyzed both the TMCA abstraction $\mathsf{TMCA}(\mathcal{P}, n, 0)$ with no concrete thread and the abstraction $\mathsf{TMCA}(\mathcal{P}, n, 1)$ with a single concrete thread. For $m = 0$, T2 proved safety in 1.2 s; for $m = 1$ in 1.5 s.

9.4.1 Current limitations

We leave a more thorough experimental investigation of termination as future work due to the following, remarkable, fact: Most existing termination proving tools oscillate between a safety proving phase (to compute supporting invariants) and a rank function synthesis phase (to compute intermediate termination arguments) [14]. Curiously, when we attempted to prove termination of further examples, several tools diverged not in their termination proving phase, but when computing supporting invariants in their safety phase. Since our TMCA abstractions for termination are similar in shape to those for safety, we suspect that this happens due to the same phenomenon that we already observed for safety in Sect. 7, i.e., bad predicate selection heuristics. Due to the tightly-knit interaction between termination provers and their backend safety prover, and the high implementation effort expected to alleviate this issue, we leave a more thorough investigation of this phenomenon for future work.

Aside: safety benchmarks

The reader is probably surprised that we are not investigating the safety benchmarks from Sect. 8 here. We have not considered these benchmarks, because they do not contain loops and are therefore trivially terminating. This is quite common in parameterized programs, where repetition is more naturally expressed through replication rather than iteration.

Still, we emphasize that to our knowledge, the termination proof we presented for the case study in this section is the first actually implemented, automated termination proof for this example.

With this, we conclude our investigation of safety and termination proofs of pararameterized programs through thread-modular counter abstraction (TMCA). As a novel framework, TMCA allows for a number of natural extensions through future work. We discuss these avenues for further research in the following section.

10 Future work

TMCA, our framework for parameterized program safety and termination, is designed to be modular and pluggable. As such, there are many directions for future work. We discuss several promising ones in this section and invite further ideas and suggestions from the community.

***k*-thread modular reasoning** Hoenicke et al. [42] investigate *k-thread modular* proofs for safety. Their method makes thread-modular proofs more expressive and is orthogonal to auxiliary state introduction. In this work, we have hinted at this possibility with the introduction of m concrete threads. It would be interesting to systematically combine k-thread modular proofs with our counter abstraction for safety. In addition, the work in [42] does not consider liveness properties. Intuitively, k-thread modularity should yield a similar increase in expressiveness for termination and liveness properties in general. An in-depth investigation of this topic makes for interesting future work.

Refinement of the thread-modular abstraction As another possibility for future work, we sketch how to further refine our thread-modular abstraction by closing the outer CEGAR loop. This corresponds to the dashed parts of Fig. 3. If the model checker reports a genuine counter-example, this may mean that the parameterized program is in fact unsafe, or that our upfront thread-modular abstraction was too coarse. If simulation on the original program finds the counter-example to be spurious, one can use predicate abstraction to refine the program's original control structure. This results in additional counters in our thread-modular abstraction. These counters are then not only capable of tracking control state, but also arbitrary predicates.

Syntactic predicate selection The interpolation abstraction approach to predicate selection is highly semantic, in that the interpolant search is left to the underlying theorem prover. While this provides a lot of freedom, it would be interesting to see how a more syntactic approach—e.g., based on syntax-guided synthesis [6]—performs.

Predicate selection for termination In Sect. 9 we conjectured that termination provers run into the same hurdles about predicate selection as safety provers, causing them to diverge on our termination benchmarks. We plan an in-depth investigation of this issue, that we expect to yield (i) a better theoretical understanding of the divergence we see, (ii) fixes to existing termination provers to make them converge on TMCA models. In close relation, we will also investigate if these changes to the safety phase of termination provers is enough, or whether tools based on *transition predicate abstraction* [64, 65] need additional extensions to the predicate discovery heuristic deployed in their termination proving phase.

General liveness properties It is well-known that fair termination is a foundational liveness property, and proofs of general linear-time properties can be reduced to checking fair termination [15, 71]. As an interesting avenue for future work, we propose to extend our termination analysis from Sect. 9 to general liveness properties. Given the practical impact of such a prover, it should be developed in lockstep with the predicate selection heuristics for termination in the previous paragraph.

Additional experiments While our safety proofs are currently limited to CHC solvers in the backend, we plan to evaluate our abstraction with further sound software verification tools as backend solvers. As we sketched above, we plan similar work on termination provers. Given our current empirical understanding of TMCA abstractions, this may lead to interesting insights and extensions of the predicate selection heuristics of these additional tools.

11 Conclusion

In this work, we present an automated, abstraction-based method for proving safety and termination of parameterized infinite-state programs. Our method cleanly separates different abstraction concerns and—in contrast to the heavy proof machinery of existing techniques—is built from a novel combination of the well-understood methods counter abstraction, thread-modular reasoning, and (transition) predicate abstraction.

In particular, for safety verification, we introduce *thread-modular counter abstraction* (TMCA), a novel abstraction method for parameterized programs (Sect. 6). Furthermore, to facilitate automation, we introduce a custom predicate selection heuristic (Sect. 7). Our implementation of both, Eldarica with TMCA [23], is freely available. Finally, we use this implementation to demonstrate the efficacy of our method on a number of benchmarks from the literature (Sect. 8).

For liveness verification, we present a generalization of TMCA to termination (Sect. 9). While not the case for thread-modular reasoning methods in general (cf. [53]), usually these techniques (e.g., [29, 40, 47])—including TMCA—are relational with respect to the environment's transition relation. That is, they over-approximate the concurrent program's *binary reachability* [17] and are thus suitable to prove termination. Similar to safety, we reduce parameterized termination to sequential termination via TMCA—the extension to general liveness properties is standard [15, 71, 72]. While we leave its systematic implementation for future work (cf. Sect. 10), we believe that this task is far more feasible than implementing previous theoretical approaches to parameterized program termination (Sect. 2).

Funding Open access funding provided by TU Wien (TUW).

References

1. Abdulla PA, Chen Y, Delzanno G, Haziza F, Hong C, Rezine A (2010) Constrained monotonic abstraction: a CEGAR for parameterized verification. In: CONCUR, Lecture notes in computer science, vol 6269. Springer, Berlin, pp 86–101
2. Abdulla PA, Cerans K, Jonsson B, Tsay Y (2000) Algorithmic analysis of programs with well quasi-ordered domains. Inf Comput 160(1–2):109–127
3. Albert E, Arenas P, Flores-Montoya A, Genaim S, Gómez-Zamalloa M, Martin-Martin E, Puebla G, Román-Díez G (2014) SACO: static analyzer for concurrent objects. In: TACAS, Lecture notes in computer science, vol 8413. Springer, Berlin, pp 562–567
4. Albert E, Flores-Montoya A, Genaim S, Martin-Martin E (2013) Termination and cost analysis of loops with concurrent interleavings. In: ATVA, Lecture notes in computer science, vol 8172. Springer, Berlin, pp 349–364
5. Alberti F, Bruttomesso R, Ghilardi S, Ranise S, Sharygina N (2012) Lazy abstraction with interpolants for arrays. In: LPAR, Lecture notes in computer science, vol 7180. Springer, Berlin, pp 46–61

6. Alur R, Bodík R, Juniwal G, Martin MMK, Raghothaman M, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2013) Syntax-guided synthesis. In: FMCAD. IEEE, pp 1–8
7. Aminof B, Rubin S, Stoilkovska I, Widder J, Zuleger F (2018) Parameterized model checking of synchronous distributed algorithms by abstraction. In: VMCAI, Lecture notes in computer science, vol 10747. Springer, Berlin, pp 1–24
8. Apt KR, de Boer FS, Olderog E (2009) Verification of sequential and concurrent programs. Texts in Computer Science. Springer, Berlin
9. Arons T, Pnueli A, Ruah S, Xu J, Zuck LD (2001) Parameterized verification with automatically computed inductive assertions. In: CAV, Lecture notes in computer science, vol 2102. Springer, Berlin, pp 221–234
10. Ball T, Podelski A, Rajamani SK (2001) Boolean and cartesian abstraction for model checking C programs. In: TACAS, Lecture notes in computer science, vol 2031. Springer, Berlin, pp 268–283
11. Benchmarks (2022) https://github.com/thpani/eldarica/tree/tmca/regression-tests/environment-abstract
12. Beyer D (2020) Advances in automatic software verification: SV-COMP 2020. In: TACAS (2), Lecture notes in computer science, vol 12079. Springer, Berlin, pp 347–367
13. Beyer D, Löwe S, Wendler P (2015) Refinement selection. In: SPIN, Lecture notes in computer science, vol 9232. Springer, Berlin, pp 20–38
14. Brockschmidt M, Cook B, Fuhs C (2013) Better termination proving through cooperation. In: CAV, Lecture notes in computer science, vol 8044. Springer, Berlin, pp 413–429
15. Cook B, Gotsman A, Podelski A, Rybalchenko A, Vardi MY (2007) Proving that programs eventually do something good. In: POPL. ACM, pp 265–276
16. Cook B, Podelski A, Rybalchenko A (2005) Abstraction refinement for termination. In: SAS, Lecture notes in computer science, vol 3672. Springer, Berlin, pp 87–101
17. Cook B, Podelski A, Rybalchenko A (2006) Termination proofs for systems code. In: PLDI. ACM, pp 415–426
18. Cook B, Podelski A, Rybalchenko A (2007) Proving thread termination. In: PLDI. ACM, pp 320–330
19. Cook B, See A, Zuleger F (2013) Ramsey vs. lexicographic termination proving. In: TACAS, Lecture notes in computer science, vol 7795. Springer, Berlin, pp 47–61
20. de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: TACAS, Lecture notes in computer science, vol 4963. Springer, Berlin, pp 337–340
21. Donaldson AF, Kaiser A, Kroening D, Wahl T (2011) Symmetry-aware predicate abstraction for shared-variable concurrent programs. In: CAV, Lecture notes in computer science, vol 6806. Springer, Berlin, pp 356–371
22. D'Silva V, Kroening D, Weissenbacher G (2008) A survey of automated techniques for formal software verification. IEEE Trans Comput Aided Des Integr Circuits Syst 27(7), 1165–1178
23. ELDARICA with TMCA (2020) https://github.com/thpani/eldarica/tree/tmca
24. Falke S, Kapur D, Sinz C (2011) Termination analysis of C programs using compiler intermediate languages. In: RTA, LIPIcs, vol 10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp 41–50
25. Farzan A, Kincaid Z, Podelski A (2014) Proofs that count. In: POPL. ACM, pp 151–164
26. Farzan A, Kincaid Z, Podelski A (2015) Proof spaces for unbounded parallelism. In: POPL. ACM, pp 407–420
27. Farzan A, Kincaid Z, Podelski A (2016) Proving liveness of parameterized programs. In: LICS. ACM, pp 185–196
28. Finkel A, Schnoebelen P (2001) Well-structured transition systems everywhere! Theor Comput Sci 256(1–2):63–92
29. Flanagan C, Qadeer S (2003) Thread-modular model checking. In: SPIN, Lecture notes in computer science, vol 2648. Springer, Berlin, pp 213–224
30. Ganjei Z, Rezine A, Eles P, Peng Z (2016) Counting dynamically synchronizing processes. STTT 18(5):517–534
31. Ganjei Z, Rezine A, Eles P, Peng Z (2015) Abstracting and counting synchronizing processes. In: VMCAI, Lecture notes in computer science, vol 8931. Springer, Berlin, pp 227–244
32. Ghilardi S, Nicolini E, Ranise S, Zucchelli D (2008) Towards SMT model checking of array-based systems. In: IJCAR, Lecture notes in computer science, vol 5195. Springer, Berlin, pp 67–82
33. Ghilardi S, Ranise S (2010) Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. Log Methods Comput Sci 6(4)
34. Giesl J, Brockschmidt M, Emmes F, Frohn F, Fuhs C, Otto C, Plücker M, Schneider-Kamp P, Ströder T, Swiderski S, Thiemann R (2014) Proving termination of programs automatically with aprove. In: IJCAR, Lecture notes in computer science, vol 8562. Springer, Berlin, pp 184–191

35. Graf S, Saïdi H (1997) Construction of abstract state graphs with PVS. In: CAV, Lecture notes in computer science, vol 1254. Springer, Berlin, pp 72–83
36. Grebenshchikov S, Lopes NP, Popeea C, Rybalchenko A (2012) Synthesizing software verifiers from proof rules. In: PLDI. ACM, pp 405–416
37. Gulwani S, Zuleger F (2010) The reachability-bound problem. In: PLDI. ACM, pp 292–304
38. Gurfinkel A, Shoham S, Meshman Y (2016) Smt-based verification of parameterized systems. In: SIGSOFT FSE. ACM, pp 338–348
39. Heizmann M, Hoenicke J, Podelski A (2013) Software model checking for people who love automata. In: CAV, Lecture notes in computer science, vol 8044. Springer, Berlin, pp 36–52
40. Henzinger TA, Jhala R, Majumdar R, Qadeer S (2003) Thread-modular abstraction refinement. In: CAV, Lecture notes in computer science, vol 2725. Springer, Berlin, pp 262–274
41. Herlihy M, Shavit N (2008) The art of multiprocessor programming. Morgan Kaufmann, Burlington
42. Hoenicke J, Majumdar R, Podelski A (2017) Thread modularity at many levels: a pearl in compositional verification. In: POPL. ACM, pp 473–485
43. Hojjat H, Rümmer P (2018) The ELDARICA horn solver. In: FMCAD. IEEE, pp 1–7
44. Hojjat H, Rümmer P, Subotic P, Yi W (2014) Horn clauses for communicating timed systems. In: HCVS, vol 169. EPTCS, pp 39–52
45. Jhala R, Majumdar R (2009) Software model checking. ACM Comput Surv 41(4):21:1–21:54
46. John A, Konnov I, Schmid U, Veith H, Widder J (2013) Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: FMCAD. IEEE, pp 201–209
47. Jones CB (1983) Specification and design of (parallel) programs. In: IFIP Congress. North-Holland/IFIP, pp 321–332
48. Kaiser A, Kroening D, Wahl T (2010) Dynamic cutoff detection in parameterized concurrent programs. In: CAV, Lecture notes in computer science, vol 6174. Springer, Berlin, pp 645–659
49. Kaiser A, Kroening D, Wahl T (2014) Lost in abstraction: monotonicity in multi-threaded programs. In: CONCUR, Lecture notes in computer science, vol 8704. Springer, Berlin, pp 141–155
50. La Torre S, Madhusudan P, Parlato G (2010) Model-checking parameterized concurrent programs using linear interfaces. In: CAV, Lecture notes in computer science, vol 6174. Springer, Berlin, pp 629–644
51. Leroux J, Rümmer P, Subotic P (2016) Guiding craig interpolation with domain-specific abstractions. Acta Inf 53(4):387–424
52. McMillan KL (2006) Lazy abstraction with interpolants. In: CAV, Lecture notes in computer science, vol 4144. Springer, Berlin, pp 123–136
53. Miné A (2011) Static analysis of run-time errors in embedded critical parallel C programs. In: ESOP, Lecture notes in computer science, vol 6602. Springer, Berlin, pp 398–418
54. Nieto LP (2001) Completeness of the owicki-gries system for parameterized parallel programs. In: IPDPS. IEEE Computer Society, p 150
55. Owicki SS (1975) Axiomatic proof techniques for parallel programs. Ph.D. thesis, Cornell University
56. Owicki SS, Gries D (1976) An axiomatic proof technique for parallel programs I. Acta Inf 6:319–340
57. Pacman (2022) https://gitlab.liu.se/live/pacman
58. Padon O, Hoenicke J, Losa G, Podelski A, Sagiv M, Shoham S (2018) Reducing liveness to safety in first-order logic. Proc ACM Program Lang 2:26:1–26:33
59. Pani T, Weissenbacher G, Zuleger F (2018) Rely-guarantee reasoning for automated bound analysis of lock-free algorithms. In: FMCAD. IEEE, pp 1–9
60. Pani T, Weissenbacher G, Zuleger F (2020) Thread-modular counter abstraction for parameterized program safety. In: FMCAD. IEEE, pp 67–76
61. Petrank E, Musuvathi M, Steensgaard B (2009) Progress guarantee for parallel programs via bounded lock-freedom. In: PLDI. ACM, pp 144–154
62. Pnueli A, Ruah S, Zuck LD (2001) Automatic deductive verification with invisible invariants. In: TACAS, Lecture notes in computer science, vol 2031. Springer, pp 82–97
63. Pnueli A, Xu J, Zuck LD (2002) Liveness with (0, 1, infty)-counter abstraction. In: CAV, Lecture notes in computer science, vol 2404. Springer, Berlin, pp 107–122
64. Podelski A, Rybalchenko A (2004) Transition invariants. In: LICS. IEEE computer society, pp 32–41
65. Podelski A, Rybalchenko A (2005) Transition predicate abstraction and fair termination. In: POPL. ACM, pp 132–144
66. Popeea C, Rybalchenko A (2012) Compositional termination proofs for multi-threaded programs. In: TACAS, Lecture notes in computer science, vol 7214. Springer, Berlin, pp 237–251
67. Rümmer P (2008) A constraint sequent calculus for first-order logic with linear integer arithmetic. In: LPAR, Lecture notes in computer science, vol 5330. Springer, Berlin, pp 274–289

68. Rümmer P, Subotic P (2013) Exploring interpolants. In: FMCAD. IEEE, pp 69–76
69. Sánchez A, Sankaranarayanan S, Sánchez C, Chang BE (2012) Invariant generation for parametrized systems using self-reflection - (extended version). In: SAS, Lecture notes in computer science, vol 7460. Springer, Berlin, pp 146–163
70. Vafeiadis V (2010) Rgsep action inference. In: VMCAI, Lecture Notes in Computer Science, vol 5944. Springer, Berlin, pp 345–361
71. Vardi MY, Wolper P (1986) An automata-theoretic approach to automatic program verification (preliminary report). In: LICS. IEEE Computer Society, pp 332–344
72. Vardi MY (1991) Verification of concurrent programs: the automata-theoretic framework. Ann Pure Appl Log 51(1–2):79–98
73. Wickerson J (2011) RGSep https://johnwickerson.github.io/talks/rely_guarantee.pdf. Imperial College lectures on Rely-Guarantee Separation Logic
74. Zuck LD, Pnueli A (2004) Model checking and abstraction to the aid of parameterized systems (a survey). Comput Lang Syst Struct 30(3–4):139–169