Thread-modular Counter Abstraction for Parameterized Program Safety

Thomas Pani
TU Wien, Vienna, Austria
pani@forsyte.at

Georg Weissenbacher
TU Wien, Vienna, Austria
weissenb@forsyte.at

Florian Zuleger
TU Wien, Vienna, Austria
zuleger@forsyte.at

***Abstract*—Automated safety proofs of parameterized software are hard: State-of-the-art methods rely on intricate abstractions and complicated proof techniques that often impede automation. We replace this heavy machinery with a clean abstraction framework built from a novel combination of *counter abstraction*, *thread-modular reasoning*, and *predicate abstraction*. Our fully automated method proves parameterized safety for a wide range of classically challenging examples in a straight-forward manner.**

***Index Terms*—parameterized program, parameterized safety, counter abstraction, thread-modular reasoning, predicate abstraction**

I. Introduction

In this paper, we present a novel method for automatically proving safety of programs that are executed by an unbounded number of concurrent threads.

Running example. Consider the program template $T[N]$ over global variables `s` and `t` and parameter N shown in Fig. 1a[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 N by n in T (Fig. 1b). We write $P(n) = P_1 \parallel \cdots \parallel P_n$ for this *parameterized program*. In this paper, we show how to automatically prove that the error location ℓ_{err} is unreachable from an initial state of $s = t = 0$ for all $n > 0$.

Despite the seemingly simple structure of the program, automatically constructing such a safety proof is hard: 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. In addition, the assertion not only refers to variables, but also to the parameter n. Thus, a safety proof for this program 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.

A. Tackling dimensions of infinity

A parameterized program – like the one above – induces 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 for sequential software, the program variables range over an infinite data domain.

State-of-the-art methods rely on heavy proof machinery to tackle these dimensions (cf. Section III). In contrast, our method is a novel combination of well-known techniques. Significantly improving the start 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:

The first step of our method, *thread-modular counter abstraction* (TMCA), deals with dimension (I) and is inspired by the well-known techniques counter abstraction [2] and thread-modular reasoning [3], [4]. TMCA uses symmetry reduction to track the number of threads in a specific local state, encodes this information in the (already infinite) data domain, and abstracts the unbounded local state into a stateless thread-modular summary. TMCA models are sequential programs that can be checked using off-the-shelf software verifiers. However, our experiments show that state-of-the-art techniques diverge on them. We thus tackle infinity dimension (II) by presenting a *novel predicate refinement heuristic* for predicate abstraction [5], [6].

II. Motivating Example

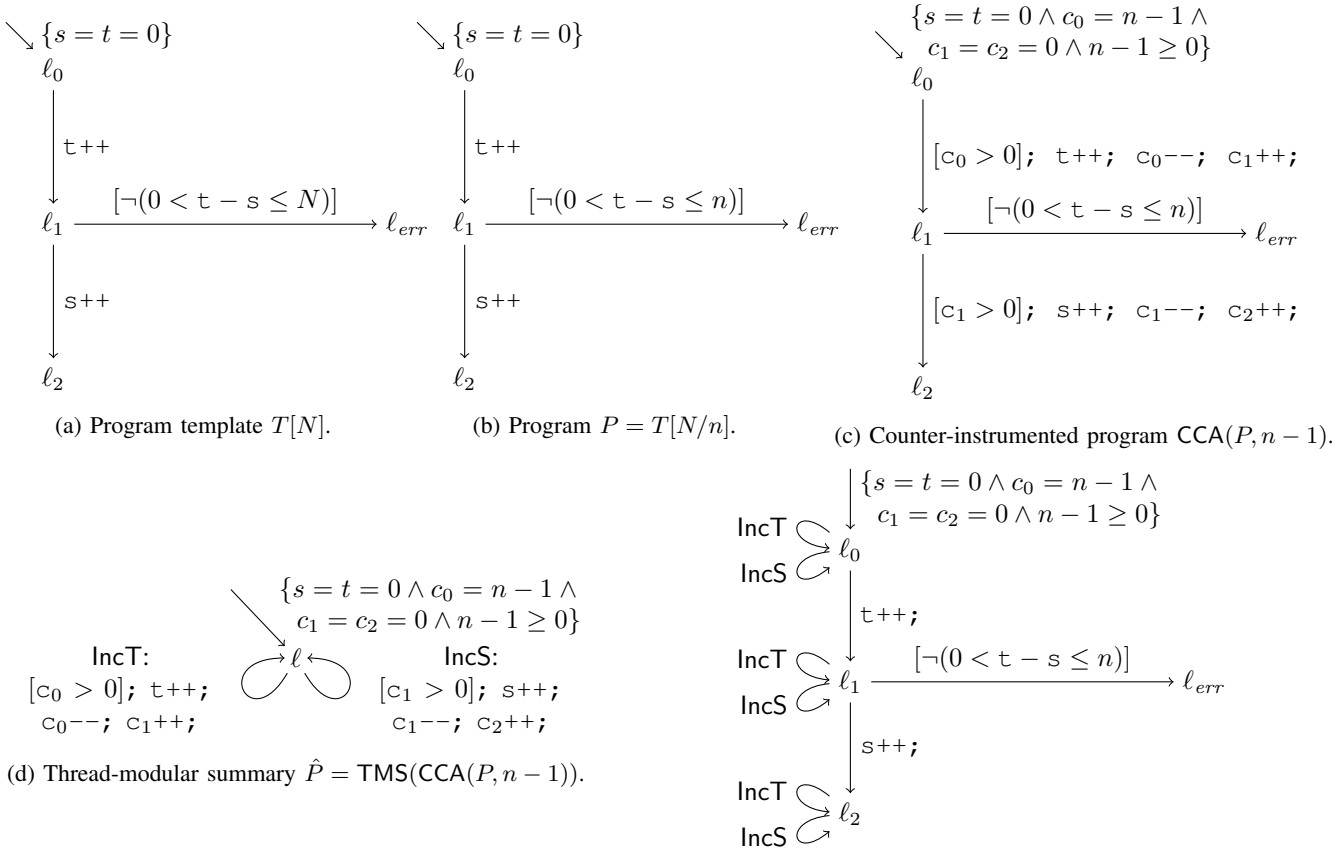
Fig. 2 gives an overview of our approach. We briefly discuss its structure and demonstrate it on our introductory example.

A. Counter instrumentation

Our method keeps one thread concrete and computes an abstraction of the $n-1$ other threads. We call these $n-1$ threads *the environment*. Our method starts by instrumenting the program $P = T[N/n]$ from Fig. 1b to track the local state of the $n-1$ environment threads in additional global counter variables. This introduction of auxiliary state serves to retain some information about the local state of all threads in the subsequent abstraction step.

Running example. In our motivating example (Fig. 1b), 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 $\{c_0, c_1, c_2\}$ and instruments the program such

[1]This slightly abstracted version of a ticket lock is adapted from the introductory example in [1]. We extend their version with an upper bounds check $s - t \leq N$. This allows us to bound $s - t$ by the number of threads n.



(a) Program template $T[N]$.

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

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

(d) Thread-modular summary $\hat{P} = \mathsf{TMS}(\mathsf{CCA}(P, n-1))$.

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

Fig. 1: Running example illustrating the thread-modular abstraction TMCA. Adapted from the introductory example in [1] by extending the assertion with an upper bounds check $s - t \leq N$ on the parameter.

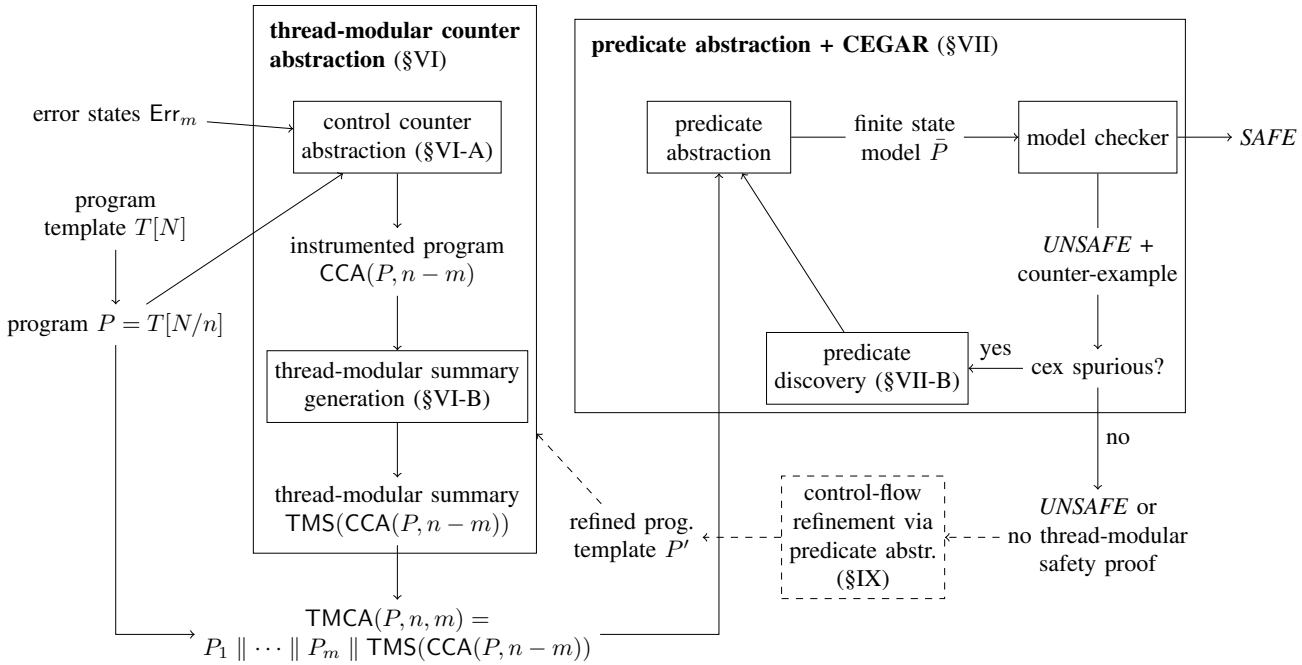


Fig. 2: Overall structure of our method. Dashed parts are beyond the scope of this work and sketched in Section IX.

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. 1c.

B. 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. In our example, the only local variable of $\mathsf{CCA}(P, n-1)$ (Fig. 1c) is the program counter. By projecting it away, we obtain $\hat{P} = \mathsf{TMS}(\mathsf{CCA}(P, n-1))$ as the thread-modular summary in Fig. 1d: Abstract transition IncT corresponds to transition $\ell_0 \to \ell_1$, while IncS corresponds to transition $\ell_1 \to \ell_2$. It is easy to see that from its initial state

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

$\hat{P}$ over-approximates the globally visible behavior of $n-1$ environment threads for all $n > 0$. Thus, instead of analyzing the parameterized program $P(n)$, we instead consider its over-approximation $\mathsf{TMCA}(T, n, 1) = P_1 \parallel \hat{P}$ (shown in Fig. 1e), where $\hat{P}$ over-approximates the behavior of $P_2 \parallel \cdots \parallel P_n$.

C. 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 (Section VIII) 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 Section VII.

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

$$c_1 < t - s \wedge t - s \leq n - c_0 \wedge \\ c_0 \geq 0 \wedge c_1 \geq 0 \wedge c_2 \geq 0 \wedge n > 0 \wedge s \geq 0 \wedge t > 0$$

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

III. 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 [7] gives an overview. In contrast, we are interested in the safety 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* [8], [9], semi-automatic *deductive techniques* [10], or those based on *small model properties* [11], [12]. In the following, we discuss the works most closely related to ours.

Ganjei et al. [13], [14] prove parameterized program safety by combining two nested CEGAR loops: Their method applies *symmetric predicate abstraction* [15], 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* [16], again abstracting the non-monotonic system into a monotonic one for which state reachability is decidable.

Kaiser et al. [17] 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 [15]. The resulting system is again non-monotonic and the authors force monotonicity as above. It is however unclear how to construct these inter-thread predicates or how to refine the monotonic abstraction.

Following a different approach, Farzan et al. [1] 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 the method is explained in theory, no implementation is given. In addition, the Petri net program model has several shortcomings. First, it is unclear how to encode a given parameterized program: even the authors present a program where "it does not seem possible to encode the verification problem for mutual exclusion by a control flow net" [1]. Second, it is unclear how to express the additional upper-bounds check on N added to our running example (Fig. 1b) given that the parameter is not symbolically represented in the control flow net.

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 our 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

[2]Synchronization mechanisms such as the *dynamic barriers* considered by Ganjei et al. [13], [14] 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.

projects away the unboundedly replicated local state. In the subsequent step, we apply *standard predicate abstraction* to deal with the infinite data domain. The discovery of counting arguments is left entirely to the predicate refinement phase. We show in Section VIII 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 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 Section IX and leave its detailed investigation for future work.

IV. 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 a superset of benchmarks compared to previous work. In particular, we make the following contributions:

1) Our framework is presented as 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 (Sections VI and VII). To our knowledge, we are the first to suggest this combination of techniques for safety 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 (Section VI). To our knowledge we are the first to propose such a counter abstraction and to apply it to parameterized programs.
   b) Predicate abstraction with standard predicate selection heuristics diverges on our abstract models (Section VIII). We present novel predicate selection heuristics to guide a CEGAR loop in the presence of these counter-abstracted summaries (Section VII).
2) We implement our method based on constrained Horn clauses (CHCs) and demonstrate its efficacy on a combined benchmark set from various sources (Section VIII).
3) The individual components of our framework lend themselves to tweaking and adaptation, 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) (Section IX).

V. Program Model and Problem Statement

In this section, we start to formally develop the technique illustrated above by formalizing our program model and problem statement.

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

$$gc : \ [cond] \mid v := e \mid gc_1; \ gc_2$$

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. We write $\nu(\mathbf{g}, \mathbf{l})$ for the valuation of global and local variables and omit its arguments wherever clear from the context. We denote by $[\![gc]\!](\nu) = \nu'$ the *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 $T[N] = (\mathit{Loc}, \delta, \ell_0, \mathit{Init})$ where Loc is a finite set of *control locations*, $\ell_0 \in \mathit{Loc}$ is the *initial location*, $\delta \subseteq \mathit{Loc} \times \mathsf{GC} \times \mathit{Loc}$ 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* $P = T[N/n] = (\mathit{Loc}, \delta', \ell_0, \mathit{Init}')$ by replacing each occurrence of N in T (i.e., in δ and Init) with the expression n. We call a pair (ℓ, ν) of a control location $\ell \in \mathit{Loc}$ and a valuation $\nu(\mathbf{g}, \mathbf{l})$ a *program state*. We represent *runs* of P as interleaved sequences of states and transitions and write $(\ell_0, \nu_0) \xrightarrow{gc_0} (\ell_1, \nu_1) \xrightarrow{gc_1} \ldots$ such that ν_0 satisfies Init', and for all $i \geq 0$ we have that $(\ell_i, gc_i, \ell_{i+1}) \in \delta'$ and $\nu_{i+1} = [\![gc_i]\!](\nu_i)$.

We define the *interleaving* of two programs $P_1 = (\mathit{Loc}_1, \delta_1, \ell_{1,0}, \mathit{Init}_1)$ and $P_2 = (\mathit{Loc}_2, \delta_2, \ell_{2,0}, \mathit{Init}_2)$ over joint global variables $\mathbf{g}$ and disjoint local variables $\mathbf{l}_1$ and $\mathbf{l}_2$ as the program $P_1 \parallel P_2 = (\mathit{Loc}_1 \times \mathit{Loc}_2, \rho, (\ell_{1,0}, \ell_{2,0}), \mathit{Init}_1 \wedge \mathit{Init}_2)$ 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 $P = (\mathit{Loc}, \delta, \ell_0, \mathit{Init})$ be a program. For *thread identifiers* $i = 1, \ldots, k$ we obtain the *instantiation* P_i of P by replacing each local variable l_j with its i-th copy $l_{j,i}$. We define the *k-times interleaving* of P as $P^k = P_1 \parallel \cdots \parallel P_k$. Finally, a program template $T[N]$ induces a *parameterized program* $P(n) = (T[N/n])^n$.

Following [18], [19], 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 vectors of global and local variables $(\mathbf{g}, \mathbf{l}_1, \ldots, \mathbf{l}_n)$. Recall that a state of $P(n)$ has the form $((\ell_1, \ldots, \ell_n), \nu)$. We define *safety* relative to a generator set of error states Err_m of $(T[N/n])^m$ for a fixed $m > 0$. $P(n)$ is *safe* iff for all $n > 0$, no run of $P(n)$ reaches an error state from the system error states Err, where

$$\begin{aligned}\mathsf{Err} \stackrel{\text{def}}{=} \{((\ell_1, \ldots, \ell_n), \nu) \ \mid \ & ((\ell_{i_1}, \ldots, \ell_{i_m}), \nu') \in \mathsf{Err}_m \text{ s.t.} \\ & \nu'(\mathbf{g}) = \nu(\mathbf{g}), \nu'(\mathbf{l}_j) = \nu(\mathbf{l}_{i_j}) \text{ for } 1 \leq j \leq m \\ & \text{and some } i_1, \ldots, i_m \text{ s.t. } 1 \leq i_1 < \cdots < i_m \leq n\}. \quad (1)\end{aligned}$$

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. 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 Err is infeasible for all parameter instantiations $n > 0$. Our method follows a two-step process that we explain in the next two sections.

VI. Tackling Infinity Dimension I: Thread-modular Counter Abstraction (TMCA)

As outlined in Section I, 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 deal with the second dimension, infinite data, in Section VII.

TMCA is inspired both by the work on *counter abstraction* [2] and *thread-modular reasoning* [3], [4]. 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 an abstraction $\hat{P}$ such that $\mathsf{TMCA}(T, n, m) = T[N/n]_1 \parallel \cdots \parallel T[N/n]_m \parallel \hat{P}$ over-approximates the reachable state space of $P(n)$, but has only finitely many control locations and variables. In the following, we explain both aspects of TMCA in further detail.

A. Control counter abstraction (CCA)

Counter abstraction [2] was introduced to abstract the parallel execution of 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 to obtain a finite-state system that is model-checked. This idea has been adapted to parameterized software [13], [17] 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.

In contrast, our method instruments counters as *auxiliary variables* [20], [21] into an *infinite-state* system: It is well-known that thread-modular reasoning is incomplete [22], but can be made more expressive by adding auxiliary state [10], [20]. 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)**.** 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 $\mathbf{g} = (g_1, \ldots, g_i)$ and control locations $Loc = \{\ell_0, \ell_1, \ldots, \ell_j\}$, let $\mathbf{g}' = (g_1, \ldots, g_i, c_0, c_1, \ldots, c_j)$. The *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

$$\ell_{src} \xrightarrow{gc'} \ell_{tgt} \in \delta' \quad \text{iff} \quad \ell_{src} \xrightarrow{gc} \ell_{tgt} \in \delta \quad \text{where}$$
$$gc' \stackrel{\text{def}}{=} [\mathsf{c}_{src} > 0];\ gc;\ \mathsf{c}_{src} := \mathsf{c}_{src} - 1;\ \mathsf{c}_{tgt} := \mathsf{c}_{tgt} + 1;$$

and $Init' \stackrel{\text{def}}{=} Init \wedge c_0 = k \wedge c_1 = \cdots = c_j = 0 \wedge k \geq 0$.

Proposition 1. *Let P be a program and let P^k be its k-times interleaving. Up to the instrumented counter variables, $\mathsf{CCA}(P, k)^k$ has the same reachable states as P^k for all $k > 0$.*

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

B. 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*. Originally conceived as an extension of Hoare logic to concurrency, *thread-modular reasoning* [3], [4] 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.

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]:

Definition 4 (Thread-modular summary)**.** 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* $\mathsf{TMS}(P) = (\{\ell\}, \delta', \ell, Init')$ for a fresh program location $\ell \notin Loc$ where $Init' \stackrel{\text{def}}{=} \exists \mathbf{l}.\, Init$ and δ' is defined as

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

Proposition 2. *Let P be a program. $\mathsf{TMS}(P)$ over-approximates the reachable global states of P's k-times interleaving P^k for all $k > 0$.*

C. Putting it together: Thread-modular counter abstr. (TMCA)

The combination of control counter abstraction (Section VI-A) and thread-modular reasoning (Section VI-B) 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.

[3]We choose this definition because it is sufficiently fine-grained for our benchmarks. In general, stronger notions of a thread-modular summary (e.g., restricting the transition relation to reachable states) can be adopted [23].

As we motivated in Section I, this is essential for constructing counting proofs. Observe the following property of the combination of CCA and TMS:

Proposition 3. *Let P be a program. Up to instrumentation variables,* $\mathsf{TMS}(\mathsf{CCA}(P,k))$ *over-approximates the reachable global states of P^k for all $k > 0$.*

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 5 (Thread-modular counter abstraction)**.** 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 control abstraction* $\mathsf{TMCA}(T,n,m)$ as the program

$$\mathsf{TMCA}(T[N],n,m) \stackrel{\text{def}}{=} \text{let } P = T[N/n] \text{ in } P_1 \parallel \cdots \parallel P_m \parallel \mathsf{TMS}(\mathsf{CCA}(P, n-m)). \quad (2)$$

Proposition 4. *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. We define R to be the set of reachable states of $P(n)$ projected to its first m components, i.e., let*

$$R = \{((\ell_1,\ldots,\ell_m),\nu(\mathbf{g},\mathbf{l}_1,\ldots,\mathbf{l}_m)) \mid \textit{s.t. } ((\ell_1,\ldots,\ell_n),\nu(\mathbf{g},\mathbf{l}_1,\ldots,\mathbf{l}_n)) \textit{ is reachable in } P(n)\}. \quad (3)$$

Then, the states reachable by $\mathsf{TMCA}(T,n,m)$ are a superset of R.

Note that by symmetry of Err, R contains an error state if and only if an error state is reachable by $P(n)$.

Theorem 1. *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 $n > 0$.*

VII. 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 Section I: the *infinite data domain*.

However, our experiments show that standard methods often fail on our models: We encode the TMCA abstraction of our benchmarks as a set of constrained Horn clauses (CHCs) [24]. Both state-of-the-art solvers ELDARICA [25] and Z3 [26] diverge on many of our examples (Table I, columns 1c and 1d; cf. Section VIII for details). We speculate that this is due to the uncommon structure of our TMCA models. In this section, we discuss how to guide a predicate abstraction-based solver to converge on TMCA models.

A. 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 [27]: 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 $A \to I$ and $I \to \neg B$. 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.

The key to converging predicate abstraction CEGAR loops is to chose the "right" interpolants. Conventional wisdom holds that referring to *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 [28], [29]. This poses a challenge for thread-modular summaries:

Running example. Recall the TMCA abstraction of our example in Fig. 1e: Due to product construction with the thread-modular summary $\mathsf{TMS}(\mathsf{CCA}(P,n-1))$, *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 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[4] (starting in an initial state where $s = t = 0$):

```
t++; IncT; IncS; [0 >= t-s];
```

The formula representing this error path is shown in Fig. 3a. If we partition the formula between IncT and IncS, an interpolating theorem prover is likely to find the new predicate $2 \le t - s$. This rules out the spurious counter-example above, but leads to another, longer one:

```
t++; IncT; IncT; IncS; IncS; [0 >= t-s];
```

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

[4]One can reproduce the behavior of this running example in the model checker ELDARICA (v2.0.2) [25] and the interpolating theorem prover PRINCESS (v2020-03-12) [30].

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

(a) Concrete interpolation query.

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

(b) Abstract interpolation query.

Fig. 3: Interpolation queries for our running example.

Instead, we want to find an invariant that relates the location counters $\texttt{c}_0$, $\texttt{c}_1$, $\texttt{c}_2$ to the values of the global variables `s` and `t`. The next section explains how to achieve this.

B. 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. *Interpolation abstraction* [31] is a state-of-the-art method to implement predicate selection. Indeed, ELDARICA with its default interpolation abstraction heuristic (Table I, column 1b) fares better than without (column 1c) but still diverges on some benchmarks. We introduce a dedicated heuristic for TMCA models to remedy this shortcoming.

Interpolation abstraction uses a set of *template terms* to abstract the interpolation query and thus guide the theorem prover in its search for an interpolant. We briefly introduce the method on our running example and refer the interested reader to the canonical description [31] for further reading.

Running example. As explained in Section I, 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

$$t = c_1 + c_2 + 1 \quad \text{and} \quad s = c_2 \quad \text{and thus} \tag{6}$$

$$t - s = (c_1 + c_2 + 1) - (c_2) = c_1 + 1 \tag{7}$$

Assume that we choose template terms $\{t - c_1, s\}$. The abstracted query is shown in Fig. 3b: Common symbols 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 Equation (7) 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 6 (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 with n in T, 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

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

We then define difference terms

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

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}.

We replace the template term heuristics of [31] with our set *Templ* but still use their search algorithm: It explores the powerset lattice $\langle \mathcal{P}(\mathit{Templ}), \subseteq \rangle$ to find the largest subsets of *Templ* for which the abstracted interpolation query is still unsat. Of these, it picks the smallest ones and computes interpolants to refine the predicate abstraction.

Intuitively, this search behavior explores relational abstractions, such as $t - c_1$, early while still allowing 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, it is worth pointing out that 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 [31].

VIII. Experiments

We implement our TMCA abstraction and predicate discovery engine [32] inside the Eldarica Horn solver [25], [31]. It 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) [24] in the standard SMT-LIB format.

Our benchmarks and results are shown in Table I. The first group of benchmarks 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 is a set of programs using unbounded thread creation taken from the software verification competition SV-COMP [33]. 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 [1]. The third group of benchmarks from [14] includes non-monotonic synchronization barriers (cf. Section III).

The columns of Table I compare the two main contributions of this work:

1) TMCA (Section VI), compared in sub-columns (1a)–(1d) to other approaches in columns (2) and (3), and
2) our predicate selection heuristic (Section VII) applied to TMCA models, compared in sub-column (1a) to other predicate selection heuristics in sub-columns (1b)–(1d).

In particular, we first compare TMCA abstraction with different backend solvers (column 1) to Pacman [14] (col. 2) and Eldarica's unbounded thread encoding[5] [18] (col. 3). 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 compile Pacman (even with the authors' help), due to outdated and commercial software dependencies. We are thus limited to citing previous results from [14] (recall from Section III that our main objective is to replace their dedicated abstraction techniques with a cleaner framework of well-established ones).

Second, we compare different backend solvers on our TMCA-abstracted models in column (1): our predicate selection heuristic from Section VII (1a), Eldarica's default heuristic [31] (1b), Eldarica without interpolation abstraction (1c) and the CHC solver in Z3 [26] (1d). Of the benchmarks, only `maximum` does not have a thread-modular proof and thus cannot be proved safe by our method. On the remaining benchmarks, our predicate selection heuristic is the only one to solve all tasks and does so well below the timeout limit of 15 minutes. Meanwhile, Eldarica with default heuristics encounters 5 timeouts, Eldarica without interpolation abstraction 10, and Z3 even 11. This shows how important an appropriate predicate discovery algorithm is for our thread-modular abstractions.

In summary, a combination of both contributions (TMCA abstraction and our predicate selection heuristic) is necessary to tackle all benchmarks.

[5]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.

IX. Future work

Our framework for parameterized program safety 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.

a) Thread-modular reasoning: [19] investigates *k-thread modular* proofs, a method orthogonal to auxiliary state introduction, to make thread-modular proofs more expressive. Another new approach to thread-modular verification is presented in [34], where a reflective abstraction is computed iteratively in a fixed point process. Integrating these approaches with our method makes an interesting area for future work.

In addition, we sketch how to further refine our thread-modular abstraction by closing the outer CEGAR loop. This corresponds to the dashed parts of Fig. 2. 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.

b) 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 [35] – performs.

c) Solving: While currently limited to CHC solvers, we plan to evaluate our abstraction with further sound software verification tools as backend solvers.

X. Conclusion

In this work, we present a method for proving parameterized safety of infinite-state programs. Our method cleanly separates different abstraction concerns and, in contrast to related work, is built from well-established methods. Finally, we demonstrated its efficacy on a number of benchmarks from the literature.

TABLE I: Benchmark results: Time to solve the respective encoding. ⏲ indicates a timeout after 15 minutes, the fastest tool for our TMCA encoding is highlighted in bold.

	(1) TMCA abstraction (Section VI)					
	(a) our heuristic	(b) ELDARICA	(c) ELDARICA	(d) Z3	(2) PACMAN	(3) ELDARICA
Benchmark	(Section VII)	`-abstract:relIneqs`	`-abstract:off`	[26]	[14]	[18]
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	⏲	⏲		⏲
fkp2014 [1]	**2.0s**	⏲	⏲	⏲		⏲
fkp2014 extd. (Fig. 1b)	**2.0s**	⏲	⏲	⏲		⏲
qw2004 [1]	**2.7s**	5.5s	⏲	⏲		⏲
locals [14]	**124.6s**	⏲	⏲	⏲	16s	⏲
shareds [14]	23.8s	**10.9s**	⏲	⏲	160s	⏲
readflag [14]	**25.5s**	⏲	⏲	⏲	34s	⏲
semaphore [14]	**36.4s**	⏲	⏲	⏲	68s	⏲
cyclic [14]	7.3s	**4.5s**	4.9s	⏲	30s	⏲
maximum [14]	—— no thread-modular proof ——				489s	⏲
parent-child [14]	—— dynamic thread creation ——				76s	dyn.thr.c.
as-many [14]	—— dynamic thread creation ——				68s	dyn.thr.c.

REFERENCES

[1] A. Farzan, Z. Kincaid, and A. Podelski, "Proofs that count," in *POPL*. ACM, 2014, pp. 151–164.

[2] A. Pnueli, J. Xu, and L. D. Zuck, "Liveness with (0, 1, infty)-counter abstraction," in *CAV*, ser. Lecture Notes in Computer Science, vol. 2404. Springer, 2002, pp. 107–122.

[3] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP Congress*. North-Holland/IFIP, 1983, pp. 321–332.

[4] C. Flanagan and S. Qadeer, "Thread-modular model checking," in *SPIN*, ser. Lecture Notes in Computer Science, vol. 2648. Springer, 2003, pp. 213–224.

[5] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *CAV*, ser. Lecture Notes in Computer Science, vol. 1254. Springer, 1997, pp. 72–83.

[6] T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and cartesian abstraction for model checking C programs," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 2031. Springer, 2001, pp. 268–283.

[7] L. D. Zuck and A. Pnueli, "Model checking and abstraction to the aid of parameterized systems (a survey)," *Comput. Lang. Syst. Struct.*, vol. 30, no. 3-4, pp. 139–169, 2004.

[8] A. Kaiser, D. Kroening, and T. Wahl, "Dynamic cutoff detection in parameterized concurrent programs," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6174. Springer, 2010, pp. 645–659.

[9] S. La Torre, P. Madhusudan, and G. Parlato, "Model-checking parameterized concurrent programs using linear interfaces," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6174. Springer, 2010, pp. 629–644.

[10] L. P. Nieto, "Completeness of the owicki-gries system for parameterized parallel programs," in *IPDPS*. IEEE Computer Society, 2001, p. 150.

[11] A. Pnueli, S. Ruah, and L. D. Zuck, "Automatic deductive verification with invisible invariants," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 2031. Springer, 2001, pp. 82–97.

[12] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck, "Parameterized verification with automatically computed inductive assertions," in *CAV*, ser. Lecture Notes in Computer Science, vol. 2102. Springer, 2001, pp. 221–234.

[13] Z. Ganjei, A. Rezine, P. Eles, and Z. Peng, "Abstracting and counting synchronizing processes," in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 8931. Springer, 2015, pp. 227–244.

[14] ——, "Counting dynamically synchronizing processes," *STTT*, vol. 18, no. 5, pp. 517–534, 2016.

[15] A. F. Donaldson, A. Kaiser, D. Kroening, and T. Wahl, "Symmetry-aware predicate abstraction for shared-variable concurrent programs," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6806. Springer, 2011, pp. 356–371.

[16] P. A. Abdulla, Y. Chen, G. Delzanno, F. Haziza, C. Hong, and A. Rezine, "Constrained monotonic abstraction: A CEGAR for parameterized verification," in *CONCUR*, ser. Lecture Notes in Computer Science, vol. 6269. Springer, 2010, pp. 86–101.

[17] A. Kaiser, D. Kroening, and T. Wahl, "Lost in abstraction: Monotonicity in multi-threaded programs," in *CONCUR*, ser. Lecture Notes in Computer Science, vol. 8704. Springer, 2014, pp. 141–155.

[18] H. Hojjat, P. Rümmer, P. Subotic, and W. Yi, "Horn clauses for communicating timed systems," in *HCVS*, ser. EPTCS, vol. 169, 2014, pp. 39–52.

[19] J. Hoenicke, R. Majumdar, and A. Podelski, "Thread modularity at many levels: a pearl in compositional verification," in *POPL*. ACM, 2017, pp. 473–485.

[20] S. S. Owicki, "Axiomatic proof techniques for parallel programs," Ph.D. dissertation, Cornell University, 1975.

[21] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Inf.*, vol. 6, pp. 319–340, 1976.

[22] K. R. Apt, F. S. de Boer, and E. Olderog, *Verification of Sequential and Concurrent Programs*, ser. Texts in Computer Science. Springer, 2009.

[23] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer, "Thread-modular abstraction refinement," in *CAV*, ser. Lecture Notes in Computer Science, vol. 2725. Springer, 2003, pp. 262–274.

[24] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*. ACM, 2012, pp. 405–416.

[25] H. Hojjat and P. Rümmer, "The ELDARICA horn solver," in *FMCAD*. IEEE, 2018, pp. 1–7.

[26] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.

[27] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV*, ser. Lecture Notes in Computer Science, vol. 4144. Springer, 2006, pp. 123–136.

[28] P. Rümmer and P. Subotic, "Exploring interpolants," in *FMCAD*. IEEE, 2013, pp. 69–76.

[29] D. Beyer, S. Löwe, and P. Wendler, "Refinement selection," in *SPIN*, ser. Lecture Notes in Computer Science, vol. 9232. Springer, 2015, pp. 20–38.

[30] P. Rümmer, "A constraint sequent calculus for first-order logic with linear integer arithmetic," in *LPAR*, ser. Lecture Notes in Computer Science, vol. 5330. Springer, 2008, pp. 274–289.

[31] J. Leroux, P. Rümmer, and P. Subotic, "Guiding craig interpolation with domain-specific abstractions," *Acta Inf.*, vol. 53, no. 4, pp. 387–424, 2016.

[32] "ELDARICA with TMCA," https://github.com/thpani/eldarica/tree/tmca, 2020.

[33] D. Beyer, "Advances in automatic software verification: SV-COMP 2020," in *TACAS (2)*, ser. Lecture Notes in Computer Science, vol. 12079. Springer, 2020, pp. 347–367.

[34] A. Sánchez, S. Sankaranarayanan, C. Sánchez, and B. E. Chang, "Invariant generation for parametrized systems using self-reflection - (extended version)," in *SAS*, ser. Lecture Notes in Computer Science, vol. 7460. Springer, 2012, pp. 146–163.

[35] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD*. IEEE, 2013, pp. 1–8.