Runtime Complexity Bounds Using Squeezers

OREN ISH-SHALOM, Tel Aviv University
SHACHAR ITZHAKY, Technion
NOAM RINETZKY and SHARON SHOHAM, Tel Aviv University

Determining upper bounds on the time complexity of a program is a fundamental problem with a variety of applications, such as performance debugging, resource certification, and compile-time optimizations. Automated techniques for cost analysis excel at bounding the resource complexity of programs that use integer values and linear arithmetic. Unfortunately, they fall short when the complexity depends more intricately on the evolution of data during execution. In such cases, state-of-the-art analyzers have shown to produce loose bounds, or even no bound at all.

We propose a novel technique that generalizes the common notion of recurrence relations based on ranking functions. Existing methods usually unfold one loop iteration and examine the resulting arithmetic relations between variables. These relations assist in establishing a recurrence that bounds the number of loop iterations. We propose a different approach, where we derive recurrences by comparing *whole traces* with *whole traces* of a lower rank, avoiding the need to analyze the complexity of intermediate states. We offer a set of global properties, defined with respect to whole traces, that facilitate such a comparison and show that these properties can be checked efficiently using a handful of local conditions. To this end, we adapt *state squeezers*, an induction mechanism previously used for verifying safety properties. We demonstrate that this technique encompasses the reasoning power of bounded unfolding, and more. We present some seemingly innocuous, yet intricate, examples that previous tools based on *cost relations* and control flow analysis fail to solve, and that our squeezer-powered approach succeeds.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Software performance;**

Additional Key Words and Phrases: Runtime complexity analysis, recurrence equations, squeezers, simulation, synthesis

ACM Reference format:
Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2022. Runtime Complexity Bounds Using Squeezers. *ACM Trans. Program. Lang. Syst.* 44, 3, Article 17 (July 2022), 36 pages.
https://doi.org/10.1145/3527632

The research leading to the presented results received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement 759102-SVIS). This research was partially supported by United States-Israel Binational Science Foundation (BSF) grants (2016260 and 2018675), Israeli Science Foundation (ISF) grants (1996/18, 1810/18, 243/19, and 2740/19), and the Pazy Foundation.
Authors' addresses: O. Ish-Shalom, N. Rinetzky, and S. Shoham, Tel Aviv University, Israel; S. Itzhaky, Technion, Haifa, Israel; emails: tuna.is.good.for.you@gmail.com, shachari@cs.technion.ac.il, maon@cs.tau.ac.il, sharon.shoham@gmail.com.

0164-0925/2022/07-ART17
https://doi.org/10.1145/3527632

1 INTRODUCTION

Cost analysis is the problem of estimating the resource usage of a given program, over all of its possible executions. It complements functional verification—of safety and liveness properties—and is an important task in formal software certification. When used in combination with functional verification, cost analysis ensures that a program is not only correct but also completes its processing in a reasonable amount of time, uses a reasonable amount of memory, communication bandwidth, and so forth. In this work, we focus on runtime complexity analysis. Although the area has been studied extensively (e.g., [3, 7, 10, 17, 19, 21, 23, 26, 35]), the general problem of constraining the number of iterations in programs containing loops with arbitrary termination conditions remains hard.

A prominent approach to computing upper bounds on the time complexity of a program identifies a well-founded numerical measure over program states that decreases in every step of the program, also called a *ranking function*. In this case, an upper bound on the measure of the initial states comprises an upper bound on the program's time complexity. Finding such measures manually is often extremely difficult. The *cost relations* approach, dating back to the work of Wegbreit [35], attempts to automate this process by using the control flow graph of the program to extract recurrence formulas that characterize this measure. Roughly speaking, the recurrences relate the measures (costs) of adjacent nodes in the graph, taking into account the cost of the step between them. In this way, the cost relations track the evolution of the measure between *every* pair of consecutive states along the executions of the program.

One limitation of cost relations is the need to capture the number of steps remaining for execution in every state—that is, all intermediate states along all executions. If the structure of the state is complex, this may require higher-order expressions, such as summing over an unbounded number of elements. As an example, consider the program in Figure 1 that implements a binary counter represented by an array of bits.

In this case, a ranking function that decreases between every two consecutive iterations of the loop, or even between two iterations that print the value of the counter, depends on the entire content of the array. Attempting to express a ranking function over the scalar variables of this program is analogous to abstracting the loop as a finite-state system that ignores the content of the array, and as such contains transition cycles (e.g., the abstract state $\langle n \mapsto n_0, i \mapsto 0\rangle$, obtained by projecting the state to the scalar variables only, repeats multiple times in any trace)—meaning that no strictly decreasing function can be defined in this way. Similarly, any attempt to consider a bounded number of bits will encounter the same difficulty.

In this article, we propose a novel approach for extracting recurrence relations capturing the time complexity of an imperative program, modeled as a transition system, by relating whole traces instead of individual states. The key idea is to relate a trace to (one or more) shorter traces. This allows to formulate a recurrence that resolves to the length of the trace and recurs over the values at the initial states only. We sidestep the need to take into account the more complex parts of the state that change along the trace (e.g., in the case of the binary counter, the array is initialized with zeros).

Our approach relies on the notion of *state squeezers* [27], previously used exclusively for the verification of safety properties. We present a novel aspect where the same squeezers can be used to determine complexity bounds, by replacing the safety property check with trace length judgments.

Squeezers provide a means to perform induction on the "size" of (initial) states to prove that all reachable states adhere to a given specification. This is accomplished by attaching *ranks* from a well-founded set to states, and defining a *squeezer function* that maps states to states of a lower rank. Note that the notion of a rank used in our work is distinct from that of a ranking function, and the two should not be confused; in particular, a rank is not required to decrease on execution

```
void binary_counter(unsigned int n) {
  unsigned int c[n];
  memset(c,0,n*sizeof(unsigned int));
  int i=0;
  while (i < n) {
    if (c[i] == 1) /*scan 1-prefix*/ { c[i] = 0; i++;           }
    else           /*increment*/     { c[i] = 1; i=0; print(c); }
  }
}
```

Fig. 1. A program that produces all combinations of n bits.

steps. Previously, squeezers were utilized for safety verification: the ability to establish safety is achieved by having the squeezer map states in a way that forms a relaxed form of a *simulation relation*, ensuring that the traces of the lower-rank states simulate the traces of the higher-rank states. Due to the simulation property, which is verified locally, safety over states with a *base* rank carries over to states of any higher rank by induction over the rank.

In this work, we use the construction of well-founded ranks and squeezers to define a *recurrence formula* representing an upper bound on the time complexity of the procedure being analyzed. We do so by expressing the complexity (length) of traces in terms of the complexity of lower-rank traces. This new setting raises additional challenges: it is no longer sufficient to relate traces to lower-rank traces; we also need to quantify the discrepancy between the lengths of the traces, as well as between their ranks. This is achieved by a certain form of simulation that is parameterized by *stuttering shapes*, which capture the discrepancy in length, and by means of a *rank bounding function*, which captures the decrease in rank. Furthermore, although our earlier work [27] limit each trace to relate to a single lower-rank trace, we have found that it is sometimes beneficial to employ a decomposition of the original trace into several consecutive *trace segments* so that each segment corresponds to some, possibly different, lower-rank trace. The segmentation simplifies the analysis of the length of the entire trace, as it creates sub-analyses that are easier to carry out, and the sum of which gives the desired recurrence formula. This also enables a richer set of recurrences to be constructed automatically, namely non-single recurrences (meaning that the recursive reference may appear more than once on the right-hand side of the equation).

The base case of the recurrence is obtained by computing an upper bound on the time complexity of base-rank states. This is typically a simpler problem that may be addressed, such as by symbolic execution due to the bounded nature of the base. The solution to the recurrence formula with the respective base case soundly overapproximates the time complexity of the procedure.[1]

We show that, conceptually, the classical approach for generating recurrences based on ranking functions can be viewed as a special case of our approach where the squeezer maps a state to its immediate successor. The real power of our approach is in the freedom to define other squeezers, producing simpler recursions, and avoiding the need for complex ranking functions.

Our use of squeezers for extracting recurrences that bound the complexity of imperative programs is related to the way analyses for functional programs (e.g., [25]) use the term(s) in recursive function calls to extract recurrences. The functional programming style coincidentally provides such candidate terms. The novelty of our approach is in introducing the concept of a squeezer explicitly, leading to a more flexible analysis because it does not restrict the squeezer to follow specific terms in the program. In particular, this allows reasoning over space in imperative programs as well.

[1]In this article, we focus on the extraction of recurrence relations rather than solving them automatically. In particular, in our experiments, recurrence relations are solved manually.

The main results of this work can be summarized as follows:

- We propose a novel technique for runtime complexity analysis of imperative programs based on state squeezers. Squeezers, together with rank bounding functions, are used for extracting recurrence relations whose solutions overapproximate the length of executions of the input program.
- We formalize the notions of *state squeezers*, *partitioned simulation*, and *rank bounding functions* that underlie the approach, and establish conditions that ensure soundness of the recurrence relations.
- We demonstrate that there are cases where compact squeezers and rank bounding functions exist, and can be verified efficiently, including cases where explicit ranking functions are too complex for existing tools.
- We implemented our approach and applied it successfully to several small but intricate programs, some of which could not have been handled by existing techniques.

A preliminary version of this work appeared in [28]. The original version can only handle deterministic programs. This extended version of that work shows how to extend the approach to non-deterministic programs. We consider two ways to do so and discuss the tradeoff between them.

2 OVERVIEW

In this section, we give a high-level description of our technique for complexity analysis using the binary counter example in Figure 1.

Example: Binary counter. The procedure in Figure 1 receives as an input a number n of bits and iterates over all their possible values in the range $0 \ldots 2^n - 1$. The "current" value is maintained in an array c that is initialized to zero and whose length is n. $c[0]$ represents the least significant bit. The loop scans the array from the least significant bit forward looking for the leftmost 0 and zeroing the prefix of 1 second. As soon as it encounters a 0, it sets it to 1 and starts the scan from the beginning. The program terminates when it reaches the end of the array ($i = n$), all array entries are zeros, and the last value was 111 . . .; at this point, all values have been enumerated.

Existing analyses. All recent methods that we are aware of (e.g., [4, 21, 25]) fail to analyze the complexity of this procedure (in fact, most methods will fail to realize that the loop terminates at all). One reason for that is the need to model the contents of the array whose size in unknown at compile time. However, even if data were modeled somehow and taken into account, finding a ranking function, which underlies existing approaches, is hard since this function is required to decrease between any two consecutive iterations along any execution. Here, for instance, to the best of our knowledge, such a function would depend on an unbounded number of elements of the array; it would need to extract the current value as an integer, along the lines of $\sum_{j=0}^{n-1} c[j] \cdot 2^j$.

The use of a ranking function for complexity analysis is somewhat analogous to the use of inductive invariants in safety verification. Both are based on induction over time along an execution. This work is inspired by previous work [27] showing that verification can also be done when the induction is performed on the size (*rank*) of the state rather than on the number of iterations, where the size of the state may correspond, for example, to the size of an unbounded data structure. We argue that similar concepts can be applied in a framework for complexity classification. In other words, we try to infer a recurrence relation that is based on the rank of the state and correlates the lengths of *complete* executions—executions that start from an initial state—of different ranks. This sidesteps the need to express the length of *partial* executions, which start from intermediate states. Although the approach applies to bounded-state systems as well, its benefits become most apparent when the program contains *a priori* unbounded stores, such as arrays.

Our approach. Roughly speaking, our approach for computing recurrence formulas that provide an upper bound on the complexity of a procedure is based on the following ingredients:

- A *rank* function $r : \mathit{init} \to X$ that maps initial states to ranks from a well-founded set $(X, \prec)$ with base B. Intuitively, the rank of the initial state governs the time complexity of the entire trace, and we also consider it to be the rank of the trace. As we shall soon see, this rank can be significantly simpler than a ranking function.
- A *squeezer* $\curlyvee : \Sigma \to \Sigma$ that maintains (some variant of) a simulation relation between states in Σ, thus ensuring a bona fide correspondence between higher-rank traces and lower-rank traces through correspondence between states.
- A *trace partition* $p_d : \Sigma \to [1..d]$ that maps each state to a segment-identifier $i \in [1..d]$, and induces a decomposition of a trace into *segments*, allowing $\curlyvee$ to map each of them to a separate, lower-rank *mini-trace*.
- A *rank bounding* function $\hat{\curlyvee} : X \times [1..d] \to X$ that provides an upper bound on the rank of the initial states of the d mini-traces based on the rank of the higher-rank trace. (The rank is not required to be uniform across mini-traces.)

All of these ingredients are synthesized automatically, as we discuss in Section 4. Next, we elaborate on each of these ingredients and illustrate them using the binary counter example. We further demonstrate how we use these ingredients to find recurrence formulas describing (an upper bound on) the complexity of the program.

Some notations. We adopt a standard encoding of a program as a transition system over a state space Σ, with a set of initial states $\mathit{init} \subseteq \Sigma$ and transition function $tr : \Sigma \to \Sigma$, where a transition corresponds to a loop iteration. We use $\mathit{reach} \subseteq \Sigma$ to denote the set of reachable states, $\mathit{reach} = \{\sigma \mid \exists \sigma_0, k.\ tr^k(\sigma_0) = \sigma \wedge \sigma_0 \in \mathit{init}\}$.

Defining the rank of a state. Ranks are taken from a well-founded set $(X, \prec)$ with a basis $B \subseteq X$ that contains all minimal elements of X. The rank function, $r : \mathit{init} \to X$, aims to abstract away irrelevant data from the (initial) state that does not effect the execution time, and only uses state "features" that do. When proper ranks are used, the rank of an initial state is all that is needed to provide a tight bound on its trace length. Since ranks are taken from a well-founded set, they can be recursed over. In the binary counter example, the chosen rank is n, namely the rank function maps each state to the size of the array. (Notice that the rank does not depend on the contents of the array; in contrast, bounding the trace length from any intermediate state, and not just initial states, would have required considering the content of the array.)

Given the rank function, our analysis extracts a recurrence formula for the complexity function $comp_x : X \to \mathbb{N} \cup \{\infty\}$ that provides an upper bound on the number of iterations of tr based on the rank of the *initial states.* In our exposition, we sometimes also refer to a time complexity function over states, $comp_s : \mathit{init} \to \mathbb{N} \cup \{\infty\}$, which is defined directly on the (initial) states, as the number of iterations in an execution that starts with some $\sigma_0 \in \mathit{init}$.

Defining a squeezer. The squeezer $\curlyvee : \Sigma \to \Sigma$ is a function that maps states to states that belong to some lower-rank traces (where the rank of a trace is determined by the rank of its initial state), down to the base ranks B. Its importance is in defining a correspondence between higher-rank traces and lower-rank ones that can be verified locally, by examining individual states rather than full traces. The kind of correspondence that the squeezer is required to ensure affects the flexibility

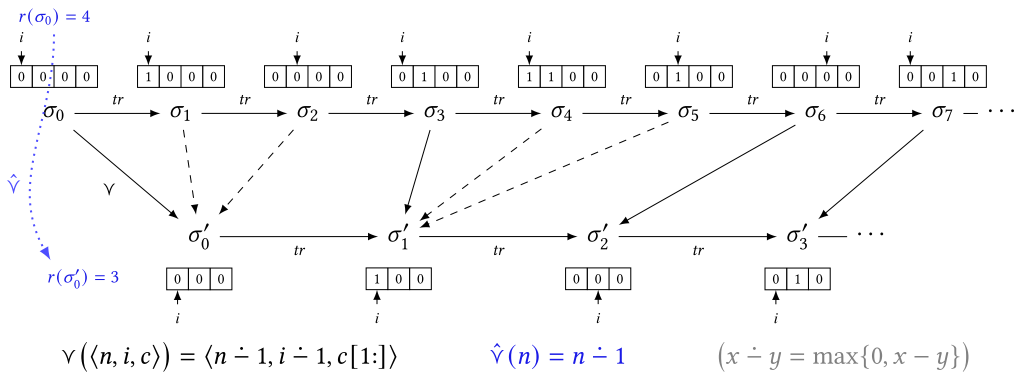


Fig. 2. Correspondence between two traces of the binary counter program. Squeezer removes the leftmost array entry, which represents the least significant bit. The rank is the array size, i.e., four on the upper trace and three on the lower one. The simulation includes only 1-,2- and 3-steps, so the length of the upper trace is at most three times that of the lower trace, yielding an overall complexity bound of $O(3^n)$.

of the approach and the kind of recurrence formulas that it may yield. To start off, consider a rather naive squeezer that satisfies the following local properties:

- Rank decrease of non-base initial states, $\sigma_0 \in init \wedge r(\sigma_0) \notin B \Rightarrow r(\curlyvee(\sigma_0)) \prec r(\sigma_0)$, and
- Simulation
  - initial anchor: $\sigma_0 \in init \Rightarrow \curlyvee(\sigma_0) \in init$,
  - stuttering-step: $\sigma \in reach \Rightarrow \exists k.\ tr(\curlyvee(\sigma)) = \curlyvee(tr^k(\sigma))$.

We refer to an instance of the stuttering-step property witnessed by a fixed k as the "k-step" property. Whenever a state σ satisfies the k-step property, we will refer to it as being $(k, 1)$-*stuttering*. (We usually only care about the smallest k that satisfies the property for a given σ.) As an example, the squeezer we consider for the binary counter program is rather intuitive: it removes the least significant bit ($c[0]$) and adjusts the index i accordingly. Doing so yields a state with rank $r(\curlyvee(\sigma_0)) = r(\sigma_0) - 1$. Figure 2 shows the correspondence between a 4-bit binary counter and a 3-bit one. The figure illustrates the simulation k-step property for $k = 1, 2, 3$: both σ_0 and σ_3 are $(3, 1)$-stuttering, σ_1 and σ_4 are $(2, 1)$-stuttering, and σ_2, σ_5, and σ_6 are $(1, 1)$-stuttering.

The simulation property induces a correlation between a higher-rank trace τ and a lower-rank one τ' such that every step of τ' is matched by k steps in τ. Now suppose that there exists some $\widehat{k} \in \mathbb{N}^+$ such that for every trace $\tau(\sigma_0)$ and every state $\sigma \in \tau(\sigma_0)$, σ is $(k, 1)$-stuttering with $1 \le k \le \widehat{k}$. This would yield the following complexity bound:

$$comp_s(\sigma_0) \le \widehat{k} \cdot comp_s(\curlyvee(\sigma_0)). \tag{1}$$

All your base.[2] What should happen if we repeatedly apply $\curlyvee$ to some initial state σ_0, each time obtaining a new, lower-rank trace? Since $r(\curlyvee(\sigma_0)) \prec r(\sigma_0)$, and since $(X, \prec)$ is well founded, we will eventually hit some state of *base rank*:

$$\curlyvee(\curlyvee(\ldots(\sigma_0))\ldots) = \sigma_0^\circ \quad \text{such that} \quad r(\sigma_0^\circ) \in B.$$

Hence, if we know the complexity of the initial states with a base rank, we can apply Equation (1) iteratively to compute an upper bound of the complexity of any initial state.

[2]https://knowyourmeme.com/memes/all-your-base-are-belong-to-us.

How many steps will be needed to get from an arbitrary initial state σ_0 to σ_0°? Clearly, this depends on the rank and the way in which $\curlyvee$ decreases it.

Consider the binary counter program again, with the rank $r(\sigma) = n$. $(\mathbb{N}, <)$ is well founded, with a single minimum 0. If we define, for example, $B = \{0, 1\}$, we know that the length of any trace with $n \in B$ is bounded by a constant, 2. (Bounding the length of traces starting from an initial state σ_0 where $r(\sigma_0) \in B$ can be done with known methods, e.g., symbolic execution.) Since the rank decreases by 1 on each "squeeze," we get the following exponential bound:

$$comp_s(\sigma_0) \leq 2 \cdot 3^{n-1} = O(3^n). \tag{2}$$

The last logical step, going from (1) to (2), is, in fact, highly involved: since Equation (1) is a mapping of *states*, solving such a recurrence for arbitrary $\curlyvee$ cannot be carried out using known automated methods. Instead, we implicitly used the rank of the state, n, to extract a recurrence over scalar values and obtain a closed-form expression. Let us make this reasoning explicit by first expressing Equation (1) in terms of $comp_x$ instead of $comp_s$:

$$comp_x(n) \leq \widehat{k} \cdot comp_x(n-1).$$

Here, $n-1$ denotes the rank obtained when squeezing an initial state of rank n. Unlike Equation (1), this is a recurrence formula over $(\mathbb{N}, <)$ that may be solved algorithmically, leading to the solution $comp_x(n) = O(3^n)$.

Surplus analysis. Assuming the worst k for all states in the trace can be too conservative—in particular, if there are only a few states that satisfy the $\widehat{k}$-step property and all of the others satisfy the 1-step property. In the latter case, if we know that at most b states in any one trace have $k > 1$, we can formulate the tighter bound:

$$comp_s(\sigma_0) \leq comp_s(\curlyvee(\sigma_0)) \;+\; \widehat{k} \cdot b. \tag{3}$$

Incidentally, in the current setting of the binary counter program, the number of $\widehat{k}$-steps (3-steps) is not bounded. So we cannot apply the inequality (3) repeatedly on any trace, as the number of 3-steps depends on the initial state. However, we can improve the analysis by partitioning the trace to two parts, as we explain next.

Segments and mini-traces. Note that both (1) and (3) "suffer" from an inherent restriction that the right-hand side contains exactly one recursive reference. As such, they are limited in expressing certain kinds of complexity classes.

To get more diverse recurrences, including recurrences with multiple recursive terms, we propose an extension of the simulation property that allows more than one lower-rank trace:

- *Partitioned* simulation:
  - initial anchor: $\sigma_0 \in init \Rightarrow \curlyvee(\sigma_0) \in init$ *(same as before)*,
  - stuttering-step: $\sigma \in reach \Rightarrow \exists k.\ tr\Big(\curlyvee(\sigma)\Big) = \curlyvee\Big(tr^k(\sigma)\Big)$ *(same as before)* <u>*or*</u>
    $\curlyvee\Big(tr(\sigma)\Big) \in init$ *(switch)*.

This definition allows a new mini-trace to start at any point along a higher-rank trace τ, thus marking the beginning of a new segment of τ. When this occurs, we call $tr(\sigma)$ a *switch state*. For the sake of uniformity, we also refer to all initial states $\sigma_0 \in init$ as switch states. Hence, each segment of τ starts with a switch state, and the mini-traces are the lower-level traces that correspond to the segments (these are the traces that start from $\curlyvee(\sigma_s)$, where σ_s is a switch state). The length of τ can now be expressed as the *sum* of lower-level mini-traces.

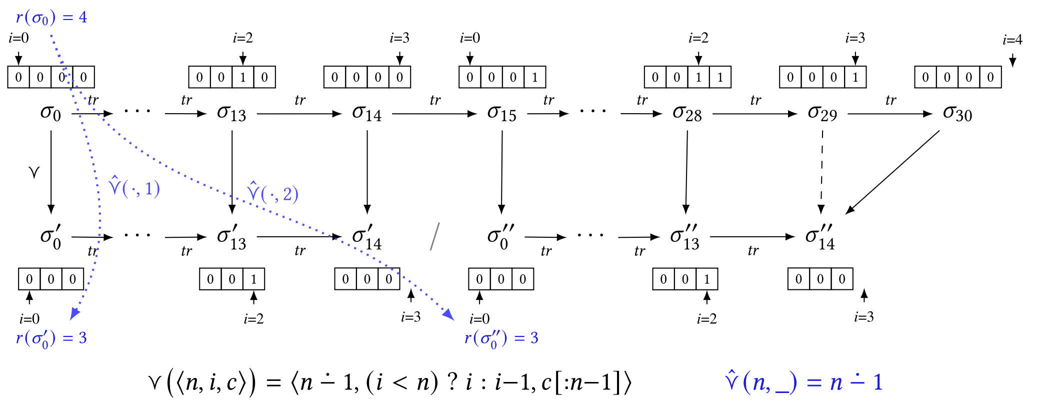


Fig. 3. An execution trace of the binary counter program that corresponds to two mini-traces of lower rank.

However, there are two problems remaining. First, we need to extend the "rank decrease of non-base initial states" requirement to any switch state to ensure that the ranks of all mini-traces are indeed lower. Namely, we need to require that if σ_s is any switch state in a trace from σ_0, then $r\big(\curlyvee(\sigma_s)\big) \prec r(\sigma_0)$. Second, even if we extend the rank decrease requirement, this definition does not suggest a way to bound the number of correlated mini-traces and their respective ranks, and therefore suggests no effective way to produce an equation for $comp_s$ as before.

To sidestep the problem of a potentially unbounded number of mini-traces, we augment the definition of simulation with a *trace partition* function; to address the challenge of the rank decrease, we use a *rank bounding* function, which is responsible both for ensuring that the rank of the mini-traces decreases and for bounding their ranks.

Defining a partition. We define a function $p_d : \Sigma \to \{1, \ldots, d\}$, parameterized by a constant d, called a *partition function*, that is weakly monotone along any trace ($p_d(\sigma) \leq p_d(tr(\sigma))$). This function induces a partition of any trace τ into (at most) d segments by grouping states based on the value of $p_d(\sigma)$. To ensure the segments and mini-traces are aligned, we require that switch states only occur at segment boundaries:

- *d-Partitioned* simulation:
  - initial anchor: $\sigma_0 \in init \Rightarrow \curlyvee(\sigma_0) \in init$ *(same as before)*,
  - k-step: $\sigma \in reach \Rightarrow \exists k.\, tr\big(\curlyvee(\sigma)\big) = \curlyvee\big(tr^k(\sigma)\big)$ *(same as before)* <u>*or*</u>
    $\curlyvee\big(tr(\sigma)\big) \in init \wedge p_d(\sigma) < p_d\big(tr(\sigma)\big)$ *(segment switch).*

In our running example, let us change $\curlyvee$ so that it shrinks the state by removing the most significant bit instead of the least. This leads to a partition of the execution trace for $r(\sigma_0) = n$ into two segments, as shown in Figure 3. The partition function is $p_d = (i \geq n \mid\mid c[n-1]) \; ? \; 2 \; : \; 1$ (essentially, $c[n-1] + 1$, except that the final state is slightly different). As can be seen from the figure, each segment simulates a mini-trace of rank $n - 1$, with $k = 1$ for all steps except for the last step (at σ_{28}) where $k = 2$. In this case, it would be folly to use the recurrence (1) with $\widehat{k} = 2$ since all steps are 1:1 except for the 2-step leaving σ_{28}. Instead, we can formulate a tighter bound:

$$comp_s(\sigma_0) \leq comp_s(\sigma'_0) + comp_s(\sigma''_0) + 2,$$

where $comp_s(\sigma'_0)$ and $comp_s(\sigma''_0)$ are the lengths of the mini-traces, and 2 is the surplus from the switch transition $\sigma_{14} \to \sigma_{15}$ plus the extra step at σ_{28}. In the case of this program, we know that $r(\sigma'_0) = r(\sigma''_0) = r(\sigma_0) - 1$, for any initial state σ_0; therefore, turning to $comp_x$, we can derive and solve the recurrence $comp_x(n) = 2 \cdot comp_x(n-1) + 2$, which together with the base yields the following bound:

$$comp_x(n) = 2^{n+1} - 2.$$

Clearly, a general condition is required to identify the ranks of the corresponding initial states of the (lower-rank) mini-traces (and at the same time ensure that they decrease).

Bounding the ranks of squeezed switch states. This is not a trivial task, because as noted previously, the squeezed ranks could be different and may depend on properties present in the corresponding switch states. To achieve this goal, once a partition function p_d is defined, we also define a rank bounding function $\hat{\curlyvee} : X \times \{1, \ldots, d\} \to X$, where for any $\sigma_0 \in init$ and switch state σ_s along the trace that starts at σ_0, $\hat{\curlyvee}$ provides a bound for the rank of $\curlyvee(\sigma_s)$ based on that of σ_0:

$$r(\curlyvee(\sigma_s)) \preceq \hat{\curlyvee}\Big(r(\sigma_0), p_d(\sigma_s)\Big) \prec r(\sigma_0). \tag{4}$$

The rightmost inequality ensures that a mini-trace that starts from $\curlyvee(\sigma_s)$ is of lower rank than σ_0, and as such extends the "rank decrease" requirement to all mini-traces. Based on this restriction, we can formulate a recurrence for $comp_x$ based on the initial rank $\rho = r(\sigma_0)$, as follows:

$$comp_x(\rho) \le \sum_{i=1}^{d} comp_x\Big(\hat{\curlyvee}(\rho, i)\Big) \;+\; (d-1) \;+\; \widehat{k} \cdot b, \tag{5}$$

where b, as before, is the number of k-steps for which $k > 1$, and $\widehat{k}$ is the bound on k ($k \le \widehat{k}$). The expression $(d-1)$ represents the transitions between segments, and $\widehat{k} \cdot b$ represents the surplus of the ρ-rank trace over the total lengths of the mini-traces.

It should be clear from the preceding definition that $\hat{\curlyvee}$ is quite intricate. How would we compute it effectively? The rank decrease of the initial states and the simulation properties were *local* by nature and thus amenable to validation with an SMT solver. The $\hat{\curlyvee}$ function is inherently *global*, defined w.r.t. an entire trace. This makes the property (4) challenging for verification methods based on SMT. To render this check more amenable to first-order reasoning, we introduce two special cases where the problem of checking (4) becomes easier: rank preservation and a single segment, explained next.

Taming $\hat{\curlyvee}$ with rank preservation. To obtain rank preservation, we extend the rank function to all states (instead of just the initial states) and require that the rank is preserved along transitions. This is appropriate in some of the scenarios we encountered. For example, the binary counter illustration satisfies the property that along any execution $\{\sigma_i\}_{i=0}^{\infty}$, the rank is preserved: $r(\sigma_i) = r(\sigma_{i+1})$. Rank preservation means that given a switch state σ_s of an arbitrary segment i, we know that $r(\sigma_s) = r(\sigma_0)$. Once this is set, $\hat{\curlyvee}$ only needs to overapproximate the rank of $\curlyvee(\sigma_s)$ in terms of the rank of the same state σ_s.

Taming $\hat{\curlyvee}$ with a single segment. In this case, checking (4) reduces to a single check of the initial state, which is the only switch state. It turns out that the restriction to a single segment is still expressive enough to handle many loop types.

Putting it all together. Theoretically, r, $\curlyvee$, p_d, and $\hat{\curlyvee}$ can be manually written by the user. However, this is a rather tedious task, which is straightforward enough to be automated. We observed that all aforementioned functions are simple enough entities that can be expressed through a strict syntax using first-order logic. Similar to our earlier work [27], we apply a generate-and-test synthesis procedure to enumerate a space of possible expressions representing them. This process is explained in Section 4.

Outline

The rest of the article is organized as follows:

- Section 3 lays out the theoretical foundations for state squeezers, their properties, and the time complexity bounds that can be obtained with them.
- Section 4 describes our approach to find appropriate state squeezers for imperative programs automatically through enumerative synthesis.
- Section 5 presents an empirical evaluation and benchmarks, comparing our implementation with state-of-the-art techniques based on cost relations.
- Section 6 shows an extension to the method that allows it to be applicable to non-deterministic programs as well. Two alternative ways to achieve this are explored.

3 COMPLEXITY ANALYSIS BASED ON SQUEEZERS

In this section, we develop the formal foundations of our approach for extracting recurrence relations describing the time complexity of an imperative program based on state squeezers. We present the ingredients that underly the approach, the conditions they are required to satisfy, and the recurrence relations they induce. In the next section, we explain how to extract the recurrences automatically. Given the recurrence relation, a dedicated (external) tool may be applied to end up with a closed formula, similar to Albert et al. [3].

We use *transition systems* to capture the semantics of a program.

Definition 3.1 (Transition Systems). A transition system is a tuple $(\Sigma, init, tr)$, where Σ is a set of *states*, $init \subseteq \Sigma$ is a set of *initial states*, and $tr : \Sigma \to \Sigma$ is a *transition function* (rather than a transition relation, since, for now, only deterministic procedures are considered). The set of *terminal states* $F \subseteq \Sigma$ is implicitly defined by $tr(\sigma) = \sigma$. An *execution trace* (or a *trace* in short) is a finite or infinite sequence of states $\tau = \sigma_0, \sigma_1, \ldots$ such that $\sigma_{i+1} = tr(\sigma_i)$ for every $0 \le i < |\tau| - 1$. A state $\sigma \in \Sigma$ defines an execution trace $\tau(\sigma) = \{tr^i(\sigma)\}_{i \in \mathbb{N}}$. Whenever there exists an index $0 \le k < |\tau|$ s.t. $\sigma_k \in F$, we truncate $\tau(\sigma)$ into a finite trace $\{tr^i(\sigma)\}_{i=0}^{k}$, where k is the minimal such index. The trace is *initial* if it starts from an initial state (i.e., $\sigma \in init$). Unless explicitly stated otherwise, all traces we consider are initial. The set of *reachable states* is $reach = \{\sigma \in \Sigma \mid \exists \sigma_0 \in init\ .\ \sigma \in \tau(\sigma_0)\}$.

Roughly, to represent a program by a transition system, we translate it into a single-loop program, where *init* consists of the states encountered when entering the loop and transitions correspond to iterations of the loop. In the case of nested loops, they are first translated to a single, flat loop by introducing an auxiliary variable that serves as program counter. This transformation is standard and straightforward, so we do not delve into it.

In the sequel, we fix a transition system $(\Sigma, init, tr)$ with a set F of terminal states and a set *reach* of reachable states.

Definition 3.2 (Complexity over States). For a state $\sigma \in \Sigma$, we denote by $comp_s(\sigma)$ the number of transitions from σ to a terminal state along $\tau(\sigma)$ (the trace that starts from σ). Formally, if $\tau(\sigma)$ does not include a terminal state (i.e., the procedure does *not* terminate from σ), then $comp_s(\sigma) = \infty$. Otherwise,

$$comp_s(\sigma) = \min\{k \in \mathbb{N} \mid tr^k(\sigma) \in F\}.$$

The complexity function of the program maps each initial state $\sigma_0 \in init$ to its time complexity $comp_s(\sigma_0) \in \mathbb{N} \cup \{\infty\}$.

Our complexity analysis derives a recurrence relation for the complexity function by expressing the length of a trace in terms of the lengths of traces that start from lower-rank states. This is achieved by (i) attaching to each initial state a *rank* from a well-founded set that we use as the argument of the complexity function and that we recur over, and (ii) defining a *squeezer* that maps each state from the original trace to a state in a lower-rank trace; the mapping forms a *partitioned simulation* according to a *partition function* that decomposes a trace to segments; each segment is simulated by a (separate) lower-rank trace, allowing to express the length of the former in terms of the latter, and, finally, (iii) defining a *rank bounding function* that expresses (an upper bound on) the ranks of the lower-rank traces in terms of the rank of the higher-rank trace. We elaborate on these components next.

3.1 Time Complexity as a Function of Rank

We start by defining a rank function that allows us to express the time complexity of an initial state by means of its rank.

Definition 3.3 (Rank). Let X be a set and $\prec$ be a well-founded partial order over X. Let $B \supseteq \min(X)$ be a *base* for X, where $\min(X)$ is the set of all minimal elements of X w.r.t. $\prec$. A *rank function* $r : init \to X$ maps each initial state to a rank in X. We extend the notion of a rank to initial traces as follows. Given an initial trace $\tau = \tau(\sigma_0)$, we define its rank to be the rank of σ_0. We refer to states σ_0 such that $r(\sigma_0) \in B$ as the *base states*. Similarly, (initial) traces whose ranks are in B are called *base traces*.

In our analysis, ranks range over $X = \mathbb{N}^m$ (for some $m \in \mathbb{N}^+$), with $\prec$ defined by the lexicographic order. Ranks let us abstract away data inside the initial execution states, which does not affect the worst-case bound on the trace length. For example, the length of traces of the binary counter program (Figure 1) is completely agnostic to the actual content of the array at the initial state. The only parameter that affects its trace length is the array size, not which integers are stored inside it. Hence, a suitable rank function in this example maps an initial state to its array length. This is despite the fact that the execution does depend on the content of the array, and, in particular, the number of remaining iterations from an intermediate state within the execution depends on it. The partial order $\prec$ and the base set B will be used to define the recurrence formula as we explain in the sequel.

We will assume from now on that $(X, \prec, B)$, as well as the rank function, are fixed, and can be understood from context. The rank function r induces a complexity function $comp_x : X \to \mathbb{N} \cup \{\infty\}$ over ranks, defined as follows.

Definition 3.4 (Complexity over Ranks). The complexity function over ranks, $comp_x : X \to \mathbb{N} \cup \{\infty\}$, is defined by

$$comp_x(\rho) = \max\{comp_s(\sigma_0) \mid r(\sigma_0) \preceq \rho \wedge \sigma_0 \in init\}.$$

The definition ensures that for every initial state $\sigma_0 \in init$, we can compute (an upper bound on) its time complexity based on its rank, as follows: $comp_s(\sigma_0) \leq comp_x(r(\sigma_0))$. The complexity of ρ takes into account all initial states with $r(\sigma) \preceq \rho$ and not only those with rank exactly ρ, to ensure monotonicity of $comp_x$ in the rank (i.e., if $\rho_1 \preceq \rho_2$, then $comp_x(\rho_1) \leq comp_x(\rho_2)$). Our approach is targeted at extracting a recurrence relation for $comp_x$.

3.2 Complexity Decomposition by Partitioned Simulation

To express the length of a trace in terms of the lengths of traces of lower ranks, we use a *squeezer* that maps states from the original trace to states of lower-rank traces and (implicitly) induces a correspondence between the original trace and the lower-rank trace(s). For now, we do not require the squeezer to decrease the rank of the trace; this requirement will be added later. The squeezer is accompanied by a partition function to form a *partitioned simulation* that allows a single higher-rank trace to be matched to multiple lower-rank traces such that their lengths may be correlated.

Definition 3.5 (Squeezer, $\curlyvee$). A squeezer is a function $\curlyvee : \Sigma \rightarrow \Sigma$.

Definition 3.6. A function $p_d : \Sigma \rightarrow \{1, \ldots, d\}$, where $d \in \mathbb{N}^+$ is called a *d-partition function* if for *every* trace $\tau = \sigma_0, \sigma_1, \ldots$ it holds that $p_d(\sigma_{i+1}) \geq p_d(\sigma_i)$ for every $0 \leq i < |\tau| - 1$.

The partition function partitions a trace into a bounded number of *segments*, where each segment consists of states with the same value of p_d. We refer to the first state of a segment as a *switch state* and to the last state of a finite segment as a *last state* (note that if τ is infinite, its last segment has no last state). In particular, this means that the initial state of a trace is a switch state, and that terminal states are last states. In other words,

- Switch states are $\mathit{init} \cup \{tr(\sigma) \mid \sigma \in \mathit{reach} \wedge p_d(\sigma) < p_d(tr(\sigma))\}$, and
- Last states are $\{\sigma \in \mathit{reach} \mid \sigma \in F \vee p_d(\sigma) < p_d(tr(\sigma))\}$.

For example, in Figure 3, σ_0 and σ_{15} are switch states and σ_{14} and σ_{30} are last states. (Note that a state may be a switch state in one trace but not in another, whereas a last state is a last state in any trace, as long as the same partition function is considered.)

Our complexity analysis requires the squeezer to form a partitioned simulation with respect to p_d. Roughly, this means that the squeezer maps each segment of a trace to a (lower-rank) trace that "simulates" it. To this end, we require all states σ within a segment of a trace to be (h, ℓ)-"stuttering," for some $h \geq 1$ and $h \geq \ell \geq 0$. Stuttering lets h consecutive transitions of σ be matched to ℓ consecutive transitions of its squeezed counterpart. If $h = \ell$, the state σ contributes to the complexity the same number of steps as the squeezed state. Otherwise, σ contributes $h - \ell$ additional steps, resulting in a longer trace. Recall that terminal states also have outgoing transitions (to themselves), but these transitions do not capture actual steps; they do not contribute to the complexity. Hence, stuttering also requires that "real" transitions of σ are matched to "real" transitions of its squeezed counterpart. This is ensured by requiring that none of the states $tr^i(\curlyvee(\sigma))$ for $i < \ell$ may be terminal states. For the last states of segments, the requirement is slightly different, as the simulation ends at the last state and a new simulation begins in the next segment. To account for the transition from the last state of one segment to the first (switch) state of the next segment, last states are considered $(1, 0)$-stuttering if they are squeezed into terminal states, unless they are terminal themselves. In any other case, they are considered $(1, 1)$-stuttering. The formal definitions follow.

Definition 3.7 (Stuttering States). Let $\curlyvee$ be a squeezer and p_d a partition function. (a) A non-last state $\sigma \in \Sigma$ is called a (h, ℓ)-*stuttering* state, for $h \geq 1$, $h \geq \ell \geq 0$, if (i) $tr^\ell(\curlyvee(\sigma)) = \curlyvee(tr^h(\sigma))$[3] and (ii) for every $i < \ell$, $tr^i(\curlyvee(\sigma)) \notin F$. (b) A last state $\sigma \in \Sigma$ is said to be $(1, 0)$-*stuttering* if $\sigma \notin F$ and $\curlyvee(\sigma) \in F$; otherwise, it is $(1, 1)$-*stuttering*.

For example, in Figure 3, all states are $(1, 1)$-stuttering with the exception of σ_{14} (a non-terminal last state that is squeezed into a terminal state) and σ_{29}, which are both $(1, 0)$-stuttering. Note that in Section 2, we refer to σ_{28} as $(2, 1)$-stuttering (and gloss over σ_{29}) since, there, to simplify the exposition, we only consider k-steps, which are a special case of (h, ℓ)-stuttering pairs where $\ell = 1$.

To obtain a partitioned simulation, switch states (along any trace), which start new segments, are further required to be squeezed into initial states (since our complexity analysis only applies to initial states). We denote by $\mathbb{S}_{p_d}(\tau)$ the switch states of trace τ according to partition p_d and by $\mathbb{S}_{p_d}$ the switch states of all traces according to the partition p_d. Namely, $\mathbb{S}_{p_d} = \mathit{init} \cup \{tr(\sigma) \mid \sigma \in \mathit{reach} \wedge p_d(\sigma) < p_d(tr(\sigma))\}$.

[3]In fact, case (a(i)) of Definition 3.7 can be relaxed to $tr^\ell(\curlyvee(\sigma)) = \curlyvee(tr^h(\sigma))$ or $tr^h(\sigma)$ is a last state; this is sound because when $tr^h(\sigma)$ is a last state, the higher-rank segment ends, so continuing the simulation is not needed.

Definition 3.8 (Partitioned Simulation). We say that a squeezer $\curlyvee : \Sigma \to \Sigma$ forms a $\{(h_i, \ell_i)\}_{i=1}^{n}$-*partitioned simulation* according to p_d, denoted $\curlyvee \sim \mathbb{PS}_{p_d}(\{(h_i, \ell_i)\}_{i=1}^{n})$, if for every non-terminal reachable state σ we have that

- σ is (h_i, ℓ_i)-stuttering for some $1 \leq i \leq n$, and
- $\sigma \in \mathbb{S}_{p_d} \Rightarrow \curlyvee(\sigma) \in init$.

When $\{(h_i, \ell_i)\}_{i=1}^{n}$ is irrelevant or clear from the context, we omit it from the notation and simply write $\curlyvee \sim \mathbb{PS}_{p_d}$.

A trace squeezed by $\curlyvee \sim \mathbb{PS}_{p_d}\left(\{(h_i, \ell_i)\}_{i=1}^{n}\right)$ may have an unbounded number of (h_i, ℓ_i)-stuttering states, which hinders the ability to define a recurrence relation based on the simulation. To overcome this, our complexity decomposition may use $\widehat{k} \geq 1$ to capture a common multiplicative factor of all stuttering pairs, with the target of leaving only a *bounded* number of states whose stuttering exceeds $\widehat{k}$ and needs to be added separately. This will become important in Theorem 3.10. Given $\widehat{k} \geq 1$, we denote by $\mathbb{E}_{\widehat{k}} \subseteq \{1, \ldots, n\}$ the set of indices such that $\frac{h_i}{\ell_i} > \widehat{k}$ (this includes indices where $\ell_i = 0$, here and elsewhere).

Observation 1 (Complexity Decomposition). *Let* $\curlyvee \sim \mathbb{PS}_{p_d}\left(\{(h_i, \ell_i)\}_{i=1}^{n}\right)$, *and* $\widehat{k} \geq 1$. *Then for every* $\sigma_0 \in init$, *we have that*

$$comp_s(\sigma_0) \leq \sum_{\sigma \in \mathbb{S}_{p_d}(\tau(\sigma_0))} \widehat{k} \cdot comp_s(\curlyvee(\sigma)) + \sum_{i \in \mathbb{E}_{\widehat{k}}} \sum_{\sigma \in \mathbb{K}_i(\tau(\sigma_0))} h_i - \ell_i \cdot \widehat{k},$$

where $\mathbb{K}_i\left(\tau(\sigma_0)\right)$ *is the multiset of* (h_i, ℓ_i)*-stuttering states in* $\tau(\sigma_0)$.

In the observation, the first addend summarizes the complexity contributed by all lower-rank traces while using $\widehat{k}$ as an upper bound on the "inflation" of the traces. However, the states that are (h_i, ℓ_i)-stuttering with $\frac{h_i}{\ell_i}$ that exceeds $\widehat{k}$ contribute additional $h_i - (\ell_i \cdot \widehat{k})$ steps to the complexity and, as a result, need to be taken into account separately. This is handled by the second addend, which adds the steps that were not accounted for by the first addend: for every such state σ', h_i steps of the higher-rank trace are matched to ℓ_i steps of the lower-rank trace. The ℓ_i steps of the lower-rank trace are already counted as part of $comp_s(\curlyvee(\sigma))$ in the first addend, where σ is the switch state at the beginning of the segment of σ'. Further, these steps are inflated by $\widehat{k}$ in the first addend. Hence, we only need to add the remaining $h_i - (\ell_i \cdot \widehat{k})$ to the length of the higher-rank trace in the second addend. Although we use the same inflation factor $\widehat{k}$ across the entire trace, a simple extension of the decomposition property may consider a different factor $\widehat{k}$ in each segment. Note that the first addend always sums over a finite number of elements since the number of switch states is at most d—the number of segments. If $\tau(\sigma_0)$ is finite, the second addend also sums over a finite number of elements.

For example, in Figure 3, $\curlyvee \sim \mathbb{PS}_{p_d}\left(\{(1,1),(1,0)\}\right)$, $\widehat{k} = 1$, and $\mathbb{E}_{\widehat{k}} = \{2\}$. The trace $\tau(\sigma_0)$ has two switch states (σ_0 and σ_{15}), and two occurrences of states (σ_{14} and σ_{29})) whose stuttering is $(1,0)$ and exceeds $\widehat{k}$. Hence, Observation 1 yields the inequality $comp_s(\sigma_0) \leq 1 \cdot comp_s(\sigma_0') + 1 \cdot comp_s(\sigma_0'') + 2 \cdot (1 - 0 \cdot 1)$, which simplifies to $comp_s(\sigma_0) \leq comp_s(\sigma_0') + comp_s(\sigma_0'') + 2$. Note that in Section 2, we obtain the same inequality but in a slightly different manner, by considering the stuttering pairs $\{(1,1),(2,1)\}$. There, the total surplus of 2 is charged to the state σ_{14} (as we do here) and to the $(2,1)$-stuttering state σ_{28}, instead of to σ_{29}—the surplus of σ_{29} is rolled over to σ_{28}.

Observation 1 considers the complexity function over states and is oblivious to the rank. In particular, it does not rely on the squeezer decreasing the rank of states. Next, we use this observation as the basis for extracting a recurrence relation for the complexity function over ranks, in which case decreasing the rank becomes important.

3.3 Extraction of Recurrence Relations over Ranks

Based on the complexity decomposition, we define recurrence relations that capture $comp_x$—the time complexity of the initial states as a function of their ranks. To go from the complexity as a function of the actual states (as in Observation 1) to the complexity as a function of their ranks, we need to express the rank of $\curlyvee(\sigma_s)$ for a switch state σ_s as a function of the rank of σ_0. To this end, we define $\hat{\curlyvee}$.

Definition 3.9. Given r, $\curlyvee$, and p_d such that $\curlyvee \sim \mathbb{PS}_{p_d}$, a function $\hat{\curlyvee} : X \times \{1, \ldots, d\} \to X$ is a *rank bounding function* if for every $\rho \in X - B$ and $1 \leq i \leq d$, if $\tau = \tau(\sigma_0)$ is an initial trace such that $r(\sigma_0) = \rho$, and $\sigma_s \in \mathbb{S}_{p_d}(\tau)$ is a switch state in τ such that $p_d(\sigma_s) = i$, the following holds:

$$\text{(i) upper bound: } r\Big(\curlyvee(\sigma_s)\Big) \preceq \hat{\curlyvee}(\rho, i) \qquad \text{and} \qquad \text{(ii) rank decrease: } \hat{\curlyvee}(\rho, i) \prec \rho.$$

In other words, Definition 3.9 requires that for every non-base initial state $\sigma_0 \in init$ and switch state σ_s at segment i of $\tau(\sigma_0)$, we have that $r(\curlyvee(\sigma_s)) \preceq \hat{\curlyvee}(r(\sigma_0), i) \prec r(\sigma_0)$. Recall that $r(\curlyvee(\sigma_s))$ is well defined since $\curlyvee(\sigma_s)$ is required to be an initial state. The definition states that $\hat{\curlyvee}(\rho, i)$ provides an upper bound on the rank of squeezed switch states in a non-base trace of rank ρ. $comp_x(r(\curlyvee(\sigma))) \leq comp_x(\hat{\curlyvee}(\rho, i))$ is ensured by the monotonicity of $comp_x$. This definition also requires the rank of non-base traces to strictly decrease when they are squeezed, as captured by the "rank decrease" inequality. Rank decrease is essential for ensuring that the extracted recurrences for $comp_x$ have a solution, as the recurrences, formally defined in Theorem 3.10, bound $comp_x(\rho)$ in terms of $comp_x(\hat{\curlyvee}(\rho, i))$; rank decrease guarantees that $\hat{\curlyvee}(\rho, i)$ is strictly smaller than ρ.

For example, in Section 2, both in the case of $d = 1$ and of $d = 2$, we can bound the ranks of the squeezed switch states via $\hat{\curlyvee}(n, i) = n \dot{-} 1$. For instance, in Figures 2 and 3, we have that $\hat{\curlyvee}(r(\sigma_0), 1) = \hat{\curlyvee}(4, 1) = 3 = r(\sigma_0') = r(\curlyvee(\sigma_0))$. In Figure 3, we have, additionally, $\hat{\curlyvee}(r(\sigma_0), 2) = \hat{\curlyvee}(4, 2) = 3 = r(\sigma_0'') = r(\curlyvee(\sigma_{15}))$.

Obtaining a rank bounding function, or even verifying that a given $\hat{\curlyvee}$ satisfies this requirement, is a challenging task. We return to this question later in this section.

These conditions allow to substitute the states for ranks in the first addend of Observation 1 and hence obtain recurrence relations for $comp_x$ over the (decreasing) ranks. To handle the second addend, we also need to bound the number of states whose stuttering, $\frac{h_i}{\ell_i}$, exceeds $\widehat{k}$. This is summarized by the following theorem.

Theorem 3.10. *Let $r : init \to X$ be a rank function, $\curlyvee : \Sigma \to \Sigma$ a squeezer, and $p_d : \Sigma \to \{1, \ldots, d\}$ a partition function such that $\curlyvee \sim \mathbb{PS}_{p_d}\Big(\{(h_i, \ell_i)\}_{i=1}^{n}\Big)$. Let $\hat{\curlyvee} : X \times \{1, \ldots, d\} \to X$ be a rank bounding function w.r.t. r, $\curlyvee$, and p_d. If, for some $\widehat{k} \geq 1$, the number of (h_i, ℓ_i)-stuttering states that appear along* any *non-base initial trace is bounded by a constant $b_i \in \mathbb{N}$ whenever $i \in \mathbb{E}_{\widehat{k}}$, then*

$$comp_x(\rho) \leq \sum_{i=1}^{d} \widehat{k} \cdot comp_x\Big(\hat{\curlyvee}(\rho, i)\Big) + \sum_{i \in \mathbb{E}_{\widehat{k}}} b_i \cdot \Big(h_i - \ell_i \cdot \widehat{k}\Big). \tag{6}$$

Note that a state may be (h_i, ℓ_i)-stuttering for several i's, in which case it is sound to count it toward any of the b_i's; in particular, we choose the one that minimizes $h_i - \ell_i \cdot \widehat{k}$.

Corollary 3.11. *Under the premises of Theorem 3.10, if $f : X \to \mathbb{N} \cup \{\infty\}$ satisfies $f(\rho) = \sum_{i=1}^{d} \widehat{k} \cdot f(\hat{\curlyvee}(\rho, i)) + \sum_{i \in \mathbb{E}_{\widehat{k}}} b_i \cdot (h_i - \ell_i \cdot \widehat{k})$ for every $\rho \in X - B$, and $comp_x(\rho) \leq f(\rho)$ for every $\rho \in B$, then $comp_x(\rho) \leq f(\rho)$ for every $\rho \in X$. We conclude that $comp_s(\sigma_0) \leq f(r(\sigma_0))$ for every $\sigma_0 \in init$.*

For example, in Section 2, we bound the length of traces of the binary counter program using two recurrence relations that take as parameter the length of the array. Formally, these recurrences are obtained as follows. In the case of $d = 1$ (Figure 2), we consider the stuttering pairs $\{(1,1),(2,1),(3,1)\}$ and choose $\widehat{k} = 3$, hence $\mathbb{E}_{\widehat{k}} = \emptyset$, leading to $f(n) = 3 \cdot f(n-1)$. In the case of $d = 2$ (Figure 3), we consider the stuttering pairs $\{(1,1),(1,0)\}$ and choose $\widehat{k} = 1$, hence $\mathbb{E}_{\widehat{k}} = \{2\}$ and the bound on the number of occurrences of $(1,0)$-stuttering states is $b_2 = 2$, leading to $f(n) = 2 \cdot f(n-1) + 2$. (As explained before, in Section 2 we slightly deviate from this formulation to simplify the presentation.)

Base-case complexity. To apply Corollary 3.11, we need to accompany Equation (6) with a bound on $comp_x(\rho)$ for the base ranks, $\rho \in B$. Fortunately, this is usually a significantly easier task. In particular, the running time of the base cases is often constant, because intuitively, the following are correlated: (a) the rank, (b) the size of the underlying data structure, and (c) the number of iterations. In this case, symbolic execution may be used to obtain bounds for base cases (as we do in our work). In essence, any method that can yield a closed-form expression for the complexity of the base cases is viable. In particular, it might be possible to apply our technique on the base case as a subproblem, using a different rank function.

3.4 Establishing the Requirements of the Recurrence Relations Extraction

Theorem 3.10 defines a recurrence relation from which an upper bound on the complexity function, $comp_x$, can be computed (Corollary 3.11). However, to ensure correctness, the premises of Theorem 3.10 must be verified. The requirement that $\curlyvee \sim \mathbb{PS}_{p_d}(\{(h_i,\ell_i)\}_{i=1}^{n})$ (see Definition 3.8) may be verified *locally* by examining individual (reachable) states: for any (reachable) state σ, the check for (h_i,ℓ_i)-stuttering and switch states can, and should, be done in tandem, and require only observing at most $\max_i h_i$ transition steps from σ and $\max_i \ell_i$ from $\curlyvee(\sigma)$. In contrast, the property required of $\hat{\curlyvee}$ is *global*: it requires $\hat{\curlyvee}(\rho, i)$ to provide an upper bound on the rank of *any* squeezed switch state that may occur in *any* position along *any* non-base initial trace whose initial state has rank ρ. Similarly, the property required of the bounds b_i is also *global*: that the number of (h_i,ℓ_i)-stuttering states along *any* non-base initial trace is at most b_i. It is therefore not clear how these requirements may be verified in general. We overcome this difficulty by imposing additional restrictions, as we discuss next.

3.4.1 Establishing Bounds on the Number of Occurrences of Stuttering States.

Bounds on the number of occurrences *per trace* that are sound *for every trace* are difficult to obtain in general. Although clever analysis methods exist that can do this kind of accounting (e.g., [23]), we found that a stronger, simpler condition applies in many cases:

- For every $\sigma \in reach$, either
  - σ is (h_i,ℓ_i)-stuttering with $\frac{h_i}{\ell_i} \le \widehat{k}$ *or*
  - σ is (h_i,ℓ_i)-stuttering (with $\frac{h_i}{\ell_i} > \widehat{k}$), *and* either σ is a *switch state* or σ is a *non-terminal last state or $tr^{h_i}(\sigma)$ is a terminal state.*

This restricts these cases to occur only at the beginnings and ends of segments. It implies a total bound of $2d \cdot \max_i(h_i - \ell_i \cdot \widehat{k})$ on the "surplus" of any trace, and therefore we substitute this expression for the rightmost sum in Equation (6). This expression assumes the "worst case" scenario where the surplus occurs both at the beginning and at the end of every segment; in some cases, this may be tightened, such as when we can verify that the surplus never occurs at the beginning of segments, in which case we may tighten the bound on the surplus to $d \cdot \max_i(h_i - \ell_i \cdot \widehat{k})$. This is the case in Figure 3. In this example, the states whose stuttering exceeds $\widehat{k} = 1$ are the state σ_{14}, which is a last state, and the state σ_{29}, for which $tr(\sigma_{29})$ is a terminal state. Both occur at the end of segments. Since both are $(1, 0)$-stuttering, we obtain a bound of $d \cdot \max_i(h_i - \ell_i \cdot \widehat{k}) = 2$ on the surplus.

More generally, it is possible to obtain the bound on the number of states whose stuttering exceeds $\widehat{k}$ by defining an auxiliary function $fuel : \Sigma \to \mathbb{N}$ that is non-increasing along every transition of the system and is required to decrease between σ and $tr^h(\sigma)$ whenever the stuttering pair, (h, ℓ), of σ exceeds $\widehat{k}$. In this case, the total bound on the "surplus" of any trace is $d \cdot \max_i(h_i - \ell_i \cdot \widehat{k})$ for $d = \max_{\sigma_0 \in init} fuel(\sigma_0)$.

3.4.2 Validating a Rank Bounding Function. The definition of a rank bounding function (Definition 3.9) encapsulates two parts. Part (ii) ensures that the rank decreases: $\hat{\curlyvee}(\rho, i) \prec \rho$ for every $\rho \in X - B$. Verifying that this requirement holds does not involve any reasoning about the states, nor traces, of the transition system. Part (i) ensures that $\hat{\curlyvee}$ provides an upper bound on the rank of squeezed switch states. Formally, it requires that $r(\curlyvee(\sigma_s)) \preceq \hat{\curlyvee}(r(\sigma_0), i)$ for every switch state σ_s in segment $i \in \{1, \ldots, d\}$ along a trace that starts from a non-base initial state σ_0. Namely, it relates the rank of the squeezed switch state, $\curlyvee(\sigma_s)$, to the rank of the initial state, σ_0, where no bound on the length of the trace between the initial state σ_0 and the switch state σ_s is known *a priori*. As such, it involves global reasoning about traces. We identify two cases in which such reasoning may be avoided: (i) the partition p_d consists of a single segment (i.e., $d = 1$), or (ii) the rank function extends to any state (and not just the initial states), while being preserved by *tr*. In both of these cases, we are able to verify the correctness of $\hat{\curlyvee}$ locally.

A single segment. In this case, the only switch state along a trace is the initial state, and hence the upper-bound requirement of $\hat{\curlyvee}$ boils down to the requirement that for every $\sigma_0 \in init$ such that $r(\sigma_0) \in X - B$, we have that $r(\curlyvee(\sigma_0)) \preceq \hat{\curlyvee}(r(\sigma_0), 1)$. This is the case in Figure 2, for instance.

LEMMA 3.12. *Let* r, $\curlyvee$, *and* $p_1 : \Sigma \to \{1\}$ *such that* $\curlyvee \sim \mathbb{PS}_{p_1}$. *Then,* $\hat{\curlyvee} : X \times \{1\} \to X$ *satisfies the upper-bound requirement of a rank bounding function if and only if* $r(\curlyvee(\sigma_0)) \preceq \hat{\curlyvee}(r(\sigma_0), 1)$ *for every* $\sigma_0 \in init$ *such that* $r(\sigma_0) \in X - B$.

Rank preservation. Another case in which the upper-bound property of $\hat{\curlyvee}$ may be verified locally is when the r can be extended to all states while being preserved by *tr*.

Definition 3.13. A function $\hat{r} : \Sigma \to X$ *extends* the rank function $r : init \to \Sigma$ if $\hat{r}$ agrees with r on the initial states—that is, $\hat{r}(\sigma_0) = r(\sigma_0)$ for every initial state $\sigma_0 \in init$. The extended rank function $\hat{r}$ is *preserved by tr* if for every reachable state σ we have that $\hat{r}(tr(\sigma)) = \hat{r}(\sigma)$.

Preservation of $\hat{r}$ by *tr* ensures that all states along an initial trace share the same rank. In particular, for a reachable switch state σ_s that lies along $\tau(\sigma_0)$, rank preservation ensures that $\hat{r}(\sigma_s) = \hat{r}(\sigma_0) = r(\sigma_0)$ (the last equality is due to the extension property), allowing us to recover the rank of σ_0 from the rank of σ_s. Therefore, the upper-bound requirement of $\hat{\curlyvee}$ simplifies into the *local* requirement that for every reachable switch state σ_s such that $\hat{r}(\sigma_s) \in X - B$, we have that $\hat{r}(\curlyvee(\sigma_s)) \preceq \hat{\curlyvee}(\hat{r}(\sigma_s), i)$, for every $i \in \{1, \ldots, d\}$.

LEMMA 3.14. *Let* r, $\curlyvee$, *and* $p_d : \Sigma \to \{1, \ldots, d\}$ *such that* $\curlyvee \sim \mathbb{PS}_{p_d}$. *Suppose that* $\hat{r} : \Sigma \to X$ *extends* r *and is preserved by tr. Then,* $\hat{\curlyvee} : X \times \{1, \ldots, d\} \to X$ *satisfies the upper-bound requirement of a rank bounding function if and only if* $\hat{r}(\curlyvee(\sigma_s)) \preceq \hat{\curlyvee}(\hat{r}(\sigma_s), i)$ *for every reachable switch state* σ_s *such that* $\hat{r}(\sigma_s) \in X - B$ *and for every* $i \in \{1, \ldots, d\}$.

For example, in Figure 3, we extend r from initial states to all states using the same definition, $\hat{r}(\langle n, i, c \rangle) = n$. Since n is not changed by the program, it is sufficient to check the upper bound property of $\hat{\curlyvee}$ locally on the switch states, which are, in the example trace, σ_0 and σ_{15}. For σ_0, the check is already local since it is an initial state. However, for σ_{15}, rank preservation allows us to check the simpler, local, property $\hat{\curlyvee}(\hat{r}(\sigma_{15}), 2) = \hat{\curlyvee}(4, 2) = 3 = \hat{r}(\curlyvee(\sigma_{15})) = \hat{r}(\sigma_0'')$, as opposed to checking $\hat{\curlyvee}(r(\sigma_0), 2) \leq r(\curlyvee(\sigma_{15}))$, where we need to track the relation between σ_0 and σ_{15}. We use this simplification to extend the verification to all traces.

Remark 1. The notion of a partitioned simulation requires a switch state σ_s to be squeezed into an initial state. This requirement may be relaxed into the requirement that σ_s is squeezed into a *reachable* state $\curlyvee(\sigma_s)$, provided that we are able to still ensure that the rank of (some) *initial* state σ_0' leading to $\curlyvee(\sigma_s)$ is smaller than the rank of the trace on which σ_s lies, and that the rank of σ_0' is properly captured by $\hat{\curlyvee}$. One case in which this is possible is when r is extended to $\hat{r}$ that is preserved by *tr*, as in this case $\hat{r}(\curlyvee(\sigma_s)) = \hat{r}(\sigma_0') = r(\sigma_0')$.

This section described *local* properties that ensure that a given program satisfies the requirements of Theorem 3.10. The locality of the properties facilitates the use of SMT solvers to perform these checks automatically. This is a key step for effective application of the method.

3.5 Trace-Length vs. State-Size Recurrences with Squeezers

A plethora of work exists for analyzing the complexity of programs (see Section 7 for a discussion of related works). Most existing techniques for automatic complexity analysis aim to find a recurrence relation on the length of the execution trace, relating the length of a trace from some state to the length of the remaining trace starting at its successor. These are recurrences on *time*, if you will, whereas our approach generates recurrences on the state *size* (captured by the rank). Is our approach completely orthogonal to preceding methods? Not quite. It turns out that from a conceptual point of view, our approach can formulate a recurrence on time as well, as we demonstrate in this section.

Obtaining trace-length recurrences based on state squeezers. The key idea is to use *tr* itself as a squeezer that squeezes each state into its immediate successor. Putting aside the initial-anchor requirement momentarily, such a squeezer forms a partitioned simulation with a single segment (i.e., $p_d \equiv 1$), in which all states along a trace are $(1, 1)$-stuttering, except for the penultimate one (if the trace is finite), which is $(1, 0)$-stuttering. Recall that squeezers must also preserve initial states (see Definition 3.8), a property that may be violated when $\curlyvee = tr$, as the successor of an initial state is not necessarily an initial state. We restore the initial-anchor property by setting $\widehat{init} = \Sigma$—that is, every state is considered an initial state.[4]

A consequence of this definition is that $comp_x$ will now provide an upper bound on the time complexity of every state and not only of the initial states, in terms of a rank that needs to be defined. If we further define a rank bounding function $\hat{\curlyvee}$, we may extract a recurrence relation of the form

$$comp_x(\rho) = comp_x(\hat{\curlyvee}(\rho)) + 1$$

(we use $\hat{\curlyvee}(\rho)$ as an abbreviation of $\hat{\curlyvee}(\rho, 1)$, since this is a special case where $d = 1$).

Defining the rank and the rank bounding function. Recall that the rank $r : \Sigma \to X$ captures the features of the (initial) states that determine the complexity. To allow maximal precision, especially since all states are now initial, we set X to be the set of *states* Σ, and define r to be the identity function, $r(\sigma) = \sigma$. With this definition, $comp_x$ and $comp_s$ become one. Next, we need to define $\prec$ and B while ensuring that $\curlyvee$ squeezes the (non-base) initial states, which are now all the states,

[4]In fact, it suffices to consider $\widehat{init} = reach$, in which case we may be able to take advantage of information from static analyses.

into states of a lower rank according to $\prec$. Since squeezers act like transitions now, having that $\curlyvee = tr$, they have the effect of decreasing the number of transitions remaining to reach a terminal state (provided that the trace is finite). We use this observation to define $\prec \subseteq \Sigma \times \Sigma$. Care is needed to ensure that $(\Sigma, \prec)$ is well founded—that is, every descending chain is finite, even though the program may not terminate. Here is the definition that achieves this goal:

$$\sigma_1 \prec \sigma_2 \iff comp_s(\sigma_1) < comp_s(\sigma_2). \tag{7}$$

Since $\curlyvee = tr$ does not decrease $comp_s$ for states that belong to infinite (non-terminating) traces ($comp_s(\curlyvee(\sigma)) = comp_s(\sigma) = \infty$, hence $\curlyvee(\sigma) \not\prec \sigma$), they must be included in B, together with the terminal states, which are minimal w.r.t. $\prec$. Namely, $B = F \cup \{\sigma \mid comp_s(\sigma) = \infty\}$. Technically, this means that the base of the recurrence needs to define $comp_x$ for these states.

The final piece in the puzzle is setting $\hat{\curlyvee} = tr$. Since $\curlyvee \sim \mathbb{PS}_{p_d}\big(\{(1,1),(1,0)\}\big)$ (when $\widehat{init} = \Sigma$), where the number of $(1,0)$-stuttering states that appear along any non-base initial trace is bounded by 1, we may use Theorem 3.10, setting $\widehat{k} = 1$, to derive the following recurrence relation, which reflects induction over time:

$$comp_x(\sigma) = comp_x(tr(\sigma)) + 1.$$

The preceding formulation represents a degenerate, naive choice of ingredients for the sake of a theoretical construction, whose purpose is to lay the foundation for a general framework that takes its strengths from both induction over time and induction over rank. This construction does not exploit the full flexibility of our framework. In particular, ranking functions obtained from termination proofs, as used in the work of Albert et al. [5], may be used to augment the rank in this setting. Further, invariants inferred from static analysis can be used to refine the recurrences.

4 SYNTHESIS

So far we have assumed that the rank function r, partition function p_d, squeezer $\curlyvee$, and a rank bounding function $\hat{\curlyvee}$ are all readily available. Clearly, they are specific to a given program. It would be too tedious for a programmer to provide these functions for the analysis of the underlying complexity. In this section, we show how to automate the process of obtaining $(r, p_d, \curlyvee, \hat{\curlyvee})$ for a class of typical looping programs. We take advantage of the fact that these components may be compact even in cases where other kinds of auxiliary functions commonly used for resource analysis, such as monotonically decreasing measures used as ranking functions, are complicated. For example, a ranking function for the binary counter program shown in Figure 1 is

$$m(n, i, c) = \left(n \cdot \sum_{j=0}^{n-1} 2^j \cdot c[j]\right) + (2^i - 1) + (n - i),$$

whereas the rank, partition, $\curlyvee$, and $\hat{\curlyvee}$ are

$$\begin{array}{ll} r(n, i, c) = n & \curlyvee(n, i, c) = \big(n - 1, (i \geq n)\ ?\ i - 1 : i, c[:n-1]\big) \\ \hat{\curlyvee}(\rho) = \rho - 1 & p_d(n, i, c) = (i \geq n \mid\mid c[n-1])\ ?\ 2 : 1. \end{array}$$

This enables the use of a relatively naive enumerative approach of multi-phase generate-and-test, employing some early pruning to discard obviously non-qualifying candidates.

4.1 SyGuS

The generation step of the synthesis loop applies syntax-guided synthesis (SyGuS [8]). Like any other SyGuS method, defining the underlying grammars is more art than science. It should be expressive enough to capture the desired terms but strict enough to effectively bound the search space. The grammars in Figures 4 and 5 describe essentially infinite spaces of expressions, but, as

$$
\begin{array}{rcl}
p_d & ::= & (\ \text{exp}\ )\ \texttt{?}\ 2\ \texttt{:}\ 1 \\
\text{exp} & ::= & \text{var} \mid \text{exp} \bowtie \text{exp} \mid \texttt{int} \\
\text{var} & ::= & \text{ID} \mid \text{ID}\ [\ \text{exp}\ ] \\
\bowtie & ::= & = \mid < \mid \leq \mid \textit{and} \mid \textit{or}
\end{array}
$$

Fig. 4. Grammar for paritition functions (p_d) generated by our SyGuS procedure.

$$
\begin{array}{rcl}
\curlyvee & ::= & \text{stmt-if} \\
\text{stmt-if} & ::= & \texttt{if}\ (\text{bexp})\ \{\text{stmt*}\}\ \texttt{else}\ \{\text{stmt*}\} \\
\text{stmt} & ::= & \text{stmt-if} \mid \text{stmt-assign} \mid \text{stmt-remove} \\
\text{stmt-assign} & ::= & \text{var} := \text{vexp}\ ; \\
\text{stmt-remove} & ::= & \texttt{remove}\ \text{arrvar}[\text{vexp}]
\end{array}
$$

$$
\begin{array}{rcl}
\hat{\curlyvee}(a, i) & ::= & (a - 1) \\
\hat{\curlyvee}((a, b), i) & ::= & (\ \text{bexp}\ )\texttt{?}\ (v, v)\ \texttt{:}\ (v, v) \\
\hat{\curlyvee}((a, b, c), i) & ::= & (\ \text{bexp}\ )\texttt{?}\ (v, v, v)\ \texttt{:}\ (v, v, v) \\
v & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid 0 \mid 1 \mid v - v \mid v \cdot v
\end{array}
$$

Fig. 5. Grammars for candidate squeezers ($\curlyvee$) and rank bounding functions ($\hat{\curlyvee}$). "vexp" denotes integer expressions, and "bexp" denotes Boolean conditions, drawn from combinations of program variables, integer constants, and arithmetic and logical operators (the specific expressions used in our experiments are described in Section 5.1).

is customary in SyGuS, a bound on the depth of the expressions is imposed. In our preliminary experiments described in Ssection 5, a bound of 2 has been used.

Ranks are taken from $\mathbb{N}^m$, where $m \in \{1, 2, 3\}$, and $\prec$ is the usual lexicographic order. The rank function r comprises one expression for each coordinate, constructed by adding/subtracting integer variables and array sizes. Boolean variables are not used in rank expressions.

Partition functions p_d. Our implementation currently supports a maximum number of two segments. This means that the partition function only assigns the values 1 and 2, and we synthesize it by generating a condition over the program's variables, *cond*, that selects between them: $p_d(\sigma) = \mathit{cond}(\sigma)\ \texttt{?}\ 2\ \texttt{:}\ 1$. Figure 4 shows the syntax of expressions used for defining partition functions.

Handling up to two segments is not an inherent limitation, but it is sufficient for our examples.

Squeezers $\curlyvee$ are the only ingredient that requires substantial synthesis effort. We represent squeezers as small loop-free imperative programs, which are natural for representing state transformations. We use a rather standard syntax with "if-then-else" and assignments, plus a `remove-adjust` operation that removes array entries and adjusts indices relating to them accordingly. We choose a minimal, loop-free fragment of the C programming language, as a natural representation of state-to-state mappings. The corresponding grammar is listed in Figure 5. An important feature of the grammar is the "remove" operation used to squeeze array stores.

Rank bounding functions $\hat{\curlyvee}$. With a well-chosen squeezer $\curlyvee$, it suffices to consider quite simple rank bounds for the mini-traces. Hence, the rank bounds defined by $\hat{\curlyvee}$ are obtained by adding, subtracting, and multiplying variables with small constants (for each coordinate of the rank). This

```
void start(uint n, uint m, bool dir) {
  uint i=m;
  while (0 < i && i < n)
  {
    if (dir) { i++; }
    else     { i--; }   }
}
```

Fig. 6. A case demonstrating that the reachability assumption is, in fact, required for successful verification of the simulation property Definition 3.7. Here, an *unreachable* state exists that violates the property. Restricting the check to reachable states (using Spacer) allows the verification to succeed.

is shown by the grammar in Figure 5. Similar to the choice of ranks, targeting simple expressions for $\hat{\curlyvee}$ helps reduce the complexity of the final recurrence that is generated from the process.

4.2 Verification

For the sake of verifying the synthesized ingredients, we fix a set $\{(h_i, \ell_i)\}_{i=1}^{n}$ of stuttering pairs and check the requirements of Theorem 3.10 as discussed in Section 3.4. In particular, we check that p_d is weakly monotone (i.e., that *cond* cannot change from true to false in any step of *tr*). Note that some of the properties may be used to discriminate some of the ingredients independent of the others. For example, the simulation requirement only depends on $\curlyvee$ and p_d. As such, it may be used to completely eliminate such candidates, for any choice of r and $\hat{\curlyvee}$. Similarly, the rank decrease property can be used to eliminate some choices of $\hat{\curlyvee}$ irrespective of the other ingredients.

Initial screening. Each candidate tuple $r, p_d, \curlyvee, \hat{\curlyvee}$ is first examined against a pool of concrete traces. If one of the properties is violated by any of the states along these traces, the candidate is discarded.

Unbounded verification. Once candidates pass the preliminary screening phase, they are verified by encoding the program and all components $r, p_d, \curlyvee, \hat{\curlyvee}$ as first-order logic expressions, and using an SMT solver (Z3 [18]) to verify that the requirements are fulfilled for all traces of the program. As mentioned in Section 3.4, all checks are local and require observing a bounded set of steps starting from a given σ. The only facet of the criteria that is difficult to encode is the fact they are required of the reachable states (and not any state). Of course, if we are able to ascertain that these are met for *all* $\sigma \in \Sigma$, including unreachable states, then the result is sound. However, for some programs and squeezers, the required properties (especially simulation) do not hold universally but are violated by unreachable states. To cope with this situation without having to manually provide invariants that capture properties of the reachable states, we use a CHC solver, Spacer [29], which is part of Z3, to check whether all reachable states in the unbounded-state system induced by the input program satisfy these properties. This can be seen as a reduction from the problem of verifying the premises of Theorem 3.10 to that of verifying a safety property.

Example 4.1. Figure 6 demonstrates that in some cases, trying to prove the simulation property for all states is indeed too strong. The following is a valid solution for $\curlyvee$ (with a single segment, $d = 1$):

$$\curlyvee \;\widehat{=}\; \texttt{if}\ (m + 1 = n)\ \{\ i = i - 1; m = m - 1; n = n - 1;\ \}\ \texttt{else}\ \{\ n = n - 1;\ \}.$$

However, the simulation property (Definition 3.7) is violated, for example, if we start from this concrete state,

$$\check{\sigma} = \big(n \mapsto 6, m \mapsto 5, dir \mapsto \text{true}, i \mapsto 1\big),$$

since $\curlyvee(tr^h(\check{\sigma})) = \big(n \mapsto 5, m \mapsto 4, dir \mapsto \text{true}, i \mapsto h\big)$, but $tr^\ell(\curlyvee(\check{\sigma})) = \big(n \mapsto 5, m \mapsto 4, dir \mapsto \text{true}, i \mapsto 0\big)$, which, for $h \geq \ell \geq 1$, are not equal. A closer look reveals that this is not a reachable state, since the following property is a loop invariant that is violated by $\check{\sigma}$:

$$dir \Rightarrow (i \geq m).$$

Spacer manages to automatically discover an appropriate invariant, which eliminates this and other spurious counterexamples, establishing the simulation property for all reachable states.

5 EMPIRICAL EVALUATION

We implemented a prototype of our complexity analyzer as a publicly available tool, SqzComp, that receives a program in a subset of C and produces recurrence relations. SqzComp is written in C++, using the Z3 C++ API [18], and using Spacer [29] via its SMTLIB2-compatible interface. For the base case of generated recurrences, we use the symbolic execution engine KLEE [12] to bound the total number of iterations by a constant.

5.1 Implementation

Our enumerative synthesis implementation prioritizes simpler expressions for all of the components, as they lead to a lighter burden on the SMT solver based checks and also contribute to simpler recurrences if they succeed. For example, prioritizing one-dimensional (scalar) ranks makes sense, since eventually the rank determines the structure of the recurrence relation. For similar reasons, we prioritize rank bounding functions $\hat{\curlyvee}$ that do not involve disjunctive bounds (i.e., conditionals). For partition functions, $p_d \equiv 1$ (no switch states) is considered first, and failing that, non-trivial partitions are tried (the search space is finite since we restrict the size of all expressions, as well as the set of constants that may be used). Following the guidelines of Lemma 3.14, a non-trivial p_d raises the need for rank preservation, which imposes further restrictions on r (which becomes, in fact, $\hat{r}$). Unlike the other ingredients, there is no clear way to prioritize the squeezer enumeration. A general rule of thumb says that the “simpler” the expressions are, the easier it will be to handle them during verification. Empirically, however, changing the enumeration order had no significant effect on the verification time.

Limitations. Our prototype implementation only handles single-loop programs where `if-then-else` statements are at the outermost level of the loop body (we manually carry out the general transformation of more complex control structures to this form).

We note that the automatic synthesis of a squeezer, rank, and rank bounding function is currently the most significant barrier for scaling the approach. In the implementation, partitioned simulations are restricted to the stuttering pairs $(1,1)$, $(1,0)$, $(2,1)$, $(3,2)$, $(4,3)$. These are relevant both for the screening phase and for the verification phase. For integer constants used in synthesized expressions, a small set of pre-defined values is considered.

Implementation Details. The implementation consists of several steps, pertaining to the steps discussed in Section 4.

Screening by random traces. To enable fast screening of false candidates $(r, p_d, \curlyvee, \hat{\curlyvee})$, we generate a set $\mathbb{P}_{init}$ of approximately 100 initial states (as we explain in the sequel). We repeatedly apply the transition function tr on each $\sigma_0 \in \mathbb{P}_{init}$, to construct $\tau(\sigma_0)$ until either it reaches a terminal state and the trace is inserted to the pool of traces, $\mathbb{P}_\tau$, or it violates some max threshold of iterations, $|\tau(\sigma_0)| > \mathbf{M}$ and is excluded from the pool (in which case, additional initial states may be added to $\mathbb{P}_{init}$).

We generate $\mathbb{P}_{init}$ by sampling random concrete states and checking whether they satisfy the initial conditions. If randomization fails to provide an order of magnitude of 100 “bona fide” initial

```
void easy(int z) {
  int x=473; int y=12;
  while (z > 0) {x=x+1;y=y-1;z=z-1;}
}
```

Fig. 7. Naive randomization fails to satisfy initial conditions of the program since hitting an initial state happens with low probability.

```
void start(int x,int y,int n,int m) {
  while (n>x) {
    if (m>y) { y = y+1; }
    else     { x = x+1; }
}}
```

Fig. 8. Distinct initial states extracted from subsequent calls to the Z3 SMT solver may appear "innocently random." In fact, they are not. All returned initial states satisfied $x < y < n < m$, thus limiting the traces pool explored from $\mathbb{P}_{init}$.

states within some maximal number of attempts, we encode the initial conditions as an SMT query and attempt to find a set of distinct initial states by repeatedly running this query, excluding the ones that are already in the pool.

Figure 7 shows an example that is bound to fail a random search for initial states (due to the use of specific initial values for the variables) but is handled easily with an SMT solver. We still favor using random sampling whenever possible, as it tends to produce a more uniform distribution of the initial states. As an example, Figure 8 shows a code example with a trivial initial condition ($init = \Sigma$), where the initial states produced by the SMT solver always satisfy $x < y < n < m$, which limits the traces in $\mathbb{P}_\tau$ to expose only some of the paths inside the loop (y is always smaller than m). Of course, real code can suffer from both limitations, which might harden the task of finding initial states. This is a generally known problem [16, 33] that is not the focus of this article.

SMT-based verification. To verify the candidates that passed the initial screening procedure, we encode the program and the properties in SMT. Our analyzer extracts the body of the loop and employs a standard translation to represent it as a transition function *tr* using SMTLIB2 operations, and constructs a representation of *init* as a Boolean formula based on the path up to and including the loop pre-header. We encode integer and Boolean variables with their corresponding SMT native sorts and use additional constraints to handle unsigned integers. As for arrays, we experimented with two different encodings: using the theory of arrays and the theory of strings (sequences). Although the theory of arrays seems like the natural choice, our squeezers may remove elements from the array (see Figure 5), which is easier to encode using string operations. In our experiments, we found that it is beneficial to restrict squeezers to only remove the first or last elements of an array, resulting in an efficient encoding using the theory of arrays. Given this encoding, the use of SMT arrays proved to be superior.

Base case. In cases where there is a small, finite number of paths (as is expected in the base case for the right rank), KLEE converges almost instantaneously. Consequently, we set a small timeout (1 second) for computing the bound on the time complexity of the base case.

5.2 Experiments

We evaluated our tool, SqzComp, on a variety of benchmark programs taken from the work of Flores-Montoya [21], as well as three additional programs: the binary counter example from

Table 1. Experimental Results

Description	Real Complexity	Inferred Bound CoFloCo	Inferred Bound SqzComp	SqzComp Time	SqzComp d
array: max value	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	< 1 sec	1
array: min value	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	< 1 sec	1
array: find first	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	< 1 sec	1
array: find last	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	< 1 sec	1
array: is-sorted	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	< 1 sec	1
array: longest asc. prefix	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	$O(\lvert A\rvert)$	< 1 sec	1
array: binary search	$O(\log(\lvert A\rvert))$	$O(\log(\lvert A\rvert))$	$O(\log(\lvert A\rvert))$	< 1 sec	1
gcd	$\max(x, y)$	$O(x + y)$	$O(x + y)$	< 1 sec	1
two-phase loop 1	$O(2n - 2x + y)$	$O(2n - 2x + y)$	$O(2n + 2y)$	< 1 sec	1
two-phase loop 2	$O(n - x + m - y)$	$O(n - x + m - y)$	$O(n - x + m - y)$	< 1 sec	1
two-phase loop 3	$O(n)$	$O(n)$	$O(n)$	< 1 sec	1
two-phase loop 4	$O(2n - x - z)$	$O(2n - x - z)$	$O(2n)$	< 1 sec	1
multi-path loop 1	$O(n)$	$O(3n)$	$O(n)$	< 1 sec	1
multi-path loop 2	$O(n)$	$O(n)$	$O(n)$	< 1 sec	1
multi-path loop 3	$O(n)$	$O(n)$	$O(n)$	< 1 sec	1
tricky init loop	$O(z)$	$O(z)$	$O(z)$	4 min	1
nested loop 1	$O(\lvert x - y\rvert)$	$O(\lvert x - y\rvert)$	$O(x + y)$	< 1 sec	1
nested loop 2	$O(a^2)$	$O(a^2)$	$O(a^2)$	16 min	1
context sensitive loop	$O(\max(n - m, m))$	$O(\max(n - m, m))$	$O(n)$	7 min	1
binary counter	$O(2^{n+1})$	∞	$O(2^{n+1})$	34 min	2
subsets	$O\left(\binom{n-m}{k}\right)$	∞	$O\left(\binom{n-m}{k}\right)$	50 min	2
monotone sequences	$O\left(\binom{n}{k}\right)$	∞	$O\left(\binom{n}{k}\right)$	50 min	2

Note: In array programs, A denotes an array. x, y, z, n, m, k, a are integer variables.

Section 2, a subsets example described in Section 5.3, and an example computing monotone sequences. These examples exhibit intricate time complexities. From the benchmark suite of Flores-Montoya [21], we filtered out non-deterministic programs, as well as programs that failed syntactic constraints that our frontend cannot currently handle. We compared SqzComp to CoFloCo [21]—a state-of-the-art tool for complexity analysis of imperative programs.

Table 1 summarizes the results of our experiments. The first column presents the name of the program, which describes its characteristics (each of the "two-phase loop" programs consists of a loop with an if statement, where the branch executed changes starting from some iteration). The second column specifies the real complexity, whereas the following two columns present the bounds inferred by SqzComp and by CoFloCo, respectively. (For SqzComp, the reported bounds are the solutions of the recurrences output by the tool.) The fourth and fifth columns respectively present the analysis running time and the number of segments used in the analysis of SqzComp.

CoFloCo's analysis time is always in the order of magnitude of 0.1 second, whether it succeeds to find a complexity bound or not. Our analysis is considerably slower, mostly due to the naive implementation of the synthesizer. The runtime of our tool is dominated by the enumerative screening phase. The time required to verify the candidates that pass the screening phase is negligible (even when Spacer is called). When both CoFloCo and SqzComp succeed, the bounds inferred by CoFloCo are sometimes tighter.

However, SqzComp manages to find tight complexity bounds for the new examples, which are not solved by CoFloCo and, to the best of our knowledge, are beyond reach of existing tools. (We also encoded the new examples as OCaml programs and ran the tool of Hoffmann et al. [25] on them, and it failed to infer bounds.)

```
1  void subsets(uint n, uint k, uint m) {
2    uint I[k]; int j = 0; bool f = true;
3    while (j >= 0) {
4      if (j >= k)     /*start left scan*/{f=false; j--;}
5      else if (j==0 && f)       /*init*/{f=true;I[0]=m;j++;}
6      else if (f)         /*right fill*/{f=true;I[j]=I[j-1]+1;j++;}
7      else if (I[j]>=n-k+j)/*left scan*/{f=false; j--;}
8      else          /*start right fill*/{f=true; I[j]=I[j]+1;j++;}
9   }}
```

```
squeezer(uint I[], uint n, uint k, uint m, int j, bool f) {
  if      (I[0]==m   &&  j>0) { m++; remove I[0]; k--; j--; }
  else if (I[0]==m)           { m++; remove I[0]; k--;      }
  else                        { m++;                        }
}
```

Fig. 9. An example program that produces all subsets of $\{m, \ldots, n-1\}$ of size k; at the bottom is the synthesized squeezer.

Although it may seem from Table 1 that the number of segments in the partition (d) determines whether the extracted complexity bound is polynomial or exponential, this is coincidental. For example, in Section 2, we obtain a bound of $O(3^n)$ for the counterexample with a single segment in the partition. In general, the complexity class inferred by our approach depends on the combination of the number of segments, the structure of the rank, the stuttering pairs, and the rank bounding functions.

5.3 Case Study: Subsets Example

This section presents one challenging example from our benchmarks, the subsets example, and the details of its complexity analysis. Notably, our method is able to infer a binomial bound, which is asymptotically tight.

The code, shown in Figure 9, iterates over all subsets of `{m,...,n-1}` of size `k`. The "current" subset is maintained in an array `I` whose length is `k`, and which is always sorted, thus avoiding generating the same set more than once. The first k iterations of the loop fill the array with values `{m,m+1,...,m+k-1}`, which represent the first subset generated. This is taken care of by the branches at lines 5 and 6 that perform a "right fill" phase, filling in the array with an ascending sequence starting from `m` at `I[0]`. Once the first k iterations are done, `j` reaches the end of the array (`j=k`) and so the next iteration will execute line 4, turning off the flag `f`, signifying that the array should now be scanned leftward. In each successive iteration, `j` is decreased, looking for the rightmost element that can be incremented. For example, if $n = 8, I = [2, 6, 7]$, this rightmost element is $I[0] = 2$. After that element is incremented, the flag `f` is turned on again, completing the "left scan" phase and starting a "right fill" phase.

A univariate recurrence. Consider the rank function $r(I, n, k, m, j, f) = n - m$, defined with respect to $(\mathbb{N}, <)$, and the squeezer shown below the program in Figure 9. The squeezer observes the first element of the array: if it is equal to m (the lower bound of the range), it removes it from the array, shrinking its size (k) by 1. It then adjusts the index j to keep pointing to the same element, unless $j = 0$, in which case that element is removed. This squeezer forms a 2-partitioned simulation, as illustrated by the traces in Figure 10. All states are $(1, 1)$-stuttering, except for σ_0, which is $(2, 1)$-stuttering, as caused by the removal of $I[0]$ when $j = 0$. The rank bounding function is $\hat{\gamma}(i, \rho) = \rho - 1$ for $i \in \{1, 2\}$. We therefore obtain the following recurrence relation:

$$comp_x(\rho) \leq 1 + comp_x(\rho - 1) + comp_x(\rho - 1).$$

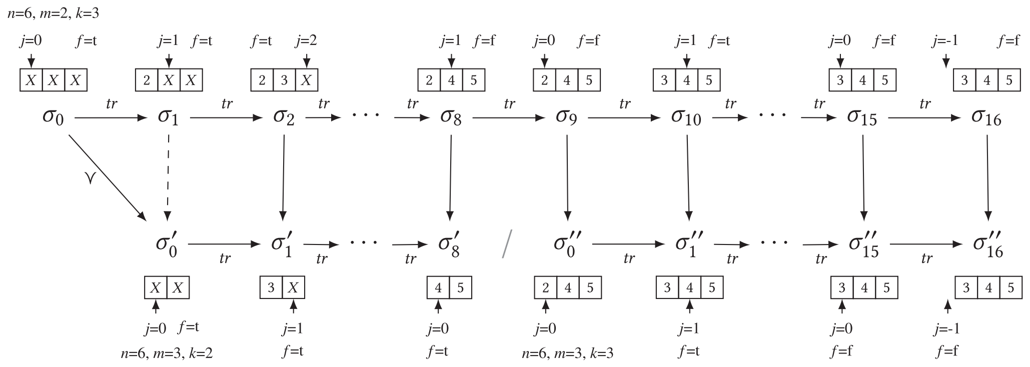


Fig. 10. An illustration of the 2-partitioned simulation for the subsets example. In the univariate case, the rank of the upper trace is $n - m$ and that of the lower traces is $n - m - 1$. In the multivariate case, the upper trace is of rank $(n - m, k)$, and lower traces of ranks $(n - m - 1, k - 1)$ and $(n - m - 1, k)$.

The base of the recurrence is $comp_x(0) = 1$, leading to the solution $comp_x(\rho) \leq 2^{\rho+1} - 1$. This means that for an initial state, $comp_s(I, n, k, m, 0, \text{true}) \leq comp_x(n - m) \leq 2^{n-m+1} - 1$.

A multivariate recurrence. Consider an alternative rank definition $r(I, n, k, m, j, f) = (n - m, k)$ defined with respect to $(\mathbb{N} \times \mathbb{N}, <)$, where "<"denotes the lexicographic order, together with the same squeezer and partition as before. The rank bounding function is now

$$\hat{\curlyvee}((\rho_1, \rho_2), i) = \begin{cases} (\rho_1 - 1, \rho_2 - 1) & i = 1 \\ (\rho_1 - 1, \rho_2) & i = 2 \end{cases}$$

The corresponding recurrence relation is

$$comp_x(\rho_1, \rho_2) \leq 1 + comp_x(\rho_1 - 1, \rho_2 - 1) + comp_x(\rho_1 - 1, \rho_2),$$

with base $comp_x(0, \_) = 1$, resulting in the solution $comp_x(\rho_1, \rho_2) \leq \binom{\rho_1+2}{\rho_2}$. In other words, for an initial state, $comp_s(I, n, k, m, 0, \text{true}) \leq comp_x(n - m, k) \leq \binom{n-m+2}{k}$.

Interestingly, this example demonstrates that the same squeezer may yield different recurrences when different ranks (and rank bounding functions) are considered. It also demonstrates a case where different segments of a trace are mapped to mini-traces of a different rank.

6 COMPLEXITY ANALYSIS FOR NON-DETERMINISTIC PROGRAMS

In this section, we extend our approach for extracting recurrence relations describing the time complexity of a program to handle non-deterministic programs. We consider two approaches. In Section 6.1, we use a transition relation rather than a transition function to model non-determinism; we adapt the definition of a partitioned simulation accordingly. In Section 6.2, we use (infinite) input arrays that store the non-deterministic choices to model non-determinism, thus reducing the complexity analysis of non-deterministic program to the analysis of deterministic programs with additional inputs. In this case, the program being analyzed is deterministic, thus its behavior is still captured by a transition function, but the transition function depends on the contents of the arrays of non-deterministic choices.

6.1 Partitioned Simulation for Transition Relations

The standard approach to modeling the semantics of non-deterministic programs as transition systems uses a transition relation to capture the transitions (steps) of the program.

Definition 6.1 (Non-deterministic Transition Systems). A non-deterministic transition system is a tuple $(\Sigma, \mathit{init}, \mathit{TR})$, where Σ and $\mathit{init} \subseteq \Sigma$ are sets of *states* and *initial states*, as in Definition 3.1, and $\mathit{TR} \subseteq \Sigma \times \Sigma$ is a *transition relation* (as opposed to a transition function). We assume that the transition relation is right-total—that is, for every $\sigma \in \Sigma$, there exists $\sigma' \in \Sigma$ such that $(\sigma, \sigma') \in \mathit{TR}$ (this may be obtained by adding self-transitions to states that have no outgoing transitions). The set of *terminal states* $F \subseteq \Sigma$ is implicitly defined by $\{\sigma \mid (\sigma, \sigma') \in \mathit{TR} \iff \sigma' = \sigma\}$—that is, the states that only transition to themselves. An *execution trace* (or a *trace* in short) is a finite or infinite sequence of states $\tau = \sigma_0, \sigma_1, \ldots$ such that $(\sigma_i, \sigma_{i+1}) \in \mathit{TR}$ for every $0 \leq i < |\tau| - 1$. We use τ_k to denote σ_k (i.e., the $k + 1$-th state along τ). When there exists an index $0 \leq k < |\tau|$ s.t. $\tau_k \in F$, we say that τ is *terminating* and we truncate τ into a finite trace $\sigma_0..\sigma_k$, where k is the minimal such index. A state $\sigma \in \Sigma$ defines a set of execution traces $T(\sigma)$ that consists of all traces that start at σ, where terminating traces are truncated as presented previously. A trace is *initial* if it starts from an initial state (i.e., $\sigma \in \mathit{init}$). Unless explicitly stated otherwise, all traces we consider are initial. The set of *reachable states* is $\mathit{reach} = \{\sigma \in \Sigma \mid \exists \sigma_0 \in \mathit{init}\,, \tau \in T(\sigma_0).\ \sigma \in \tau\}$.

The definition of the complexity (Definition 3.2) is adapted accordingly to take into account all execution traces of a state.

Definition 6.2 (Complexity of Non-deterministic Programs). For a state $\sigma \in \Sigma$, we denote by $\mathit{comp}_s(\sigma)$ the maximal number of transitions that can be executed from σ before a terminal state is encountered. Formally, if there exists $\tau \in T(\sigma)$ that does not include a terminal state (i.e., the procedure has an execution that does *not* terminate from σ), then $\mathit{comp}_s(\sigma) = \infty$. Otherwise,

$$\mathit{comp}_s(\sigma) = \max_{\tau \in T(\sigma)} \min_{k \in \mathbb{N}} \{\tau_k \in F\}.$$

As before, the complexity function of the program maps each initial state $\sigma_0 \in \mathit{init}$ to its time complexity $\mathit{comp}_s(\sigma_0) \in \mathbb{N} \cup \{\infty\}$.

The definition of complexity over ranks (Definition 3.4) carries over and requires no adaptation.

To use squeezers as a mechanism for comparing high-rank traces to low-rank traces via a partitioned simulation, we need to take into account the property that a state in a non-deterministic transition system may have multiple outgoing traces. In this case, we need to ensure that every outgoing trace τ of a high rank state is properly matched with *some* outgoing trace τ' of the corresponding low-rank state. Note that we do not need to require a matching with every outgoing trace of the lower-rank state, since the complexity of the lower-rank state considers the maximum over all traces, including τ'. Since we are also interested in keeping track of the degree of stuttering (which determines the relation between the lengths of the traces and accordingly the complexities of the states), we first define a notion of stuttering traces. Roughly, a (higher-rank) trace of length h is ℓ-stuttering if there exists a (lower-rank) trace of length ℓ such that the first and last state of the traces are matched by the squeezer. We further require that the lower-rank trace does not reach a terminal state unless the higher-rank trace does.

Definition 6.3 (Stuttering Traces). An h-step trace is a sequence $\sigma_0..\sigma_h$ of $h + 1$ states such that $\langle \sigma_i, \sigma_{i+1} \rangle \in \mathit{TR}$ for every $0 \leq i < h$. Given a squeezer $\curlyvee$, an h-step trace is ℓ-stuttering if there is an ℓ-step trace $\sigma'_0..\sigma'_\ell$ such that (i) $\sigma'_0 = \curlyvee(\sigma_0)$ and $\sigma'_\ell = \curlyvee(\sigma'_h)$, and (ii) for all $i < \ell$, $\sigma'_i \notin F$. A (finite or infinite) trace τ is (h, ℓ)-stuttering if it has a prefix of length h that is ℓ-stuttering.

Next, we refine the notion of stuttering states (Definition 3.7) to refer to a given trace on which the state resides. Last states require special treatment as before. Note that due to non-determinism, a state may be a last state in one trace but not in another.

Definition 6.4 (Stuttering States in a Trace). Let $\curlyvee$ be a squeezer and p_d a partition function. Let τ be a trace and $\sigma \in \Sigma$ a state that resides on τ. Denote by τ^σ the suffix of τ that starts from σ. (a) If σ is a non-last state in τ, then it is called (h, ℓ)-*stuttering in* τ, for $|\tau^\sigma| \geq h \geq 1$, $h \geq \ell \geq 0$, if τ^σ is (h, ℓ)-stuttering; (b) if σ is a last state in τ, then it is said to be $(1, 0)$-*stuttering in* τ if $\sigma \notin F$ and $\curlyvee(\sigma) \in F$, and otherwise it is $(1, 1)$-*stuttering in* τ.

Note that when the transition relation *TR* is in fact a function, Definition 6.4 collapses into Definition 3.7.

Having adapted the definition of stuttering states to be relative to a trace, we now adapt the definition of a partitioned simulation (Definition 3.8) to require that every outgoing trace of σ admits one of the stuttering behaviors in $\{(h_i, \ell_i)\}_{i=1}^n$.

Definition 6.5 (Partitioned Simulation). We say that a squeezer $\curlyvee : \Sigma \to \Sigma$ forms a $\{(h_i, \ell_i)\}_{i=1}^n$-*partitioned simulation* according to p_d, denoted $\curlyvee \sim \mathbb{PS}_{p_d}\big(\{(h_i, \ell_i)\}_{i=1}^n\big)$, if for every non-terminal reachable state σ and every outgoing trace τ of σ we have that

- σ is (h_i, ℓ_i)-stuttering in τ for some $1 \leq i \leq n$, and
- $\sigma \in \mathbb{S}_{p_d} \Rightarrow \curlyvee(\sigma) \in init$.

The decomposition observation (Observation 1), which provides the foundation for our analysis, is refined to provide a bound on the length of each initial trace (as opposed to the complexity of an initial state, which corresponds to the maximum over all of its outgoing traces). The bound for trace τ is given in terms of the complexities of lower-rank states that match the switch states along τ. Each trace may have different switch states, but their number is at most d (when a d-partition is considered). The decomposition observation is later used to provide a bound on the complexity of the initial states.

Observation 2 (Complexity Decomposition). *Let* $\curlyvee \sim \mathbb{PS}_{p_d}\Big(\{(h_i, \ell_i)\}_{i=1}^n\Big)$, *and* $\widehat{k} \geq 1$. *Let* $\mathbb{E}_{\widehat{k}} \subseteq \{1, \ldots, n\}$ *be the set of indices such that* $\frac{h_i}{\ell_i} > \widehat{k}$ *(including indices where* $\ell_i = 0$*). Then for every initial trace* τ*, we have that*

$$|\tau| \leq \sum_{\sigma \in \mathbb{S}_{p_d}(\tau)} \widehat{k} \cdot comp_s(\curlyvee(\sigma)) + \sum_{i \in \mathbb{E}_{\widehat{k}}} \sum_{\sigma \in \mathbb{K}_i(\tau)} h_i - \ell_i \cdot \widehat{k},$$

where $\mathbb{S}_{p_d}(\tau)$ *is the set of switch states in* τ *and* $\mathbb{K}_i\big(\tau\big)$ *is the multiset of* (h_i, ℓ_i)*-stuttering states in* τ*.*

As in the deterministic case, to make the leap from the complexity decomposition to a recurrence relation that captures the complexity by means of the rank, we use a rank bounding function. The requirement of a rank bounding function refers to all initial traces, as before (Definition 3.9); the only difference is that an initial state no longer induces a unique initial trace $\tau(\sigma_0)$ but may be the source of multiple initial traces in $T(\sigma_0)$:

Definition 6.6. Given r, $\curlyvee$, and p_d such that $\curlyvee \sim \mathbb{PS}_{p_d}$, a function $\hat{\curlyvee} : X \times \{1, \ldots, d\} \to X$ is a *rank bounding function* if for every $\rho \in X - B$ and $1 \leq i \leq d$, if $\tau \in T(\sigma_0)$ is an initial trace such that $r(\sigma_0) = \rho$, and $\sigma_s \in \mathbb{S}_{p_d}(\tau)$ is a switch state in τ such that $p_d(\sigma_s) = i$, the following holds:

$$\text{(i) upper bound: } r\Big(\curlyvee(\sigma_s)\Big) \leq \hat{\curlyvee}(\rho, i) \quad \text{and} \quad \text{(ii) rank decrease: } \hat{\curlyvee}(\rho, i) \prec \rho.$$

With the adapted definitions, Theorem 3.10 and Corollary 3.11 are preserved, allowing us to extract recurrence relations for non-deterministic programs.

Establishing the Required Properties. We now describe how to extend the results of Section 3.4 to the non-deterministic setting.

Simulation property. With TR being a relation, we can no longer map a given $\sigma \in \Sigma$ to a single outgoing trace and accordingly to a single state $tr^{h_i}(\sigma)$ as in Section 3.7. Similarly, we cannot use $tr^{\ell_i}\big(\curlyvee(\sigma)\big)$. Instead, we have to use Definitions 6.3 and 6.4, which mention trace prefixes of lengths h_i and ℓ_i starting at σ and $\curlyvee(\sigma)$, respectively. Validating the property that a state is (h_i, ℓ_i)-stuttering in τ for one of the pairs in $\{(h_i, \ell_i)\}_i$ means introducing state variables for the intermediate states along sequences, and quantifying over them, as captured by the following formula. For uniformity of the presentation we use σ_0 to denote the state σ for which the stuttering requirement is formulated. We further denote $h = \max_i h_i$ and $\ell = \max_i \ell_i$:

$$\begin{aligned}&\forall \sigma_1 \cdots \sigma_h. \bigwedge_{0 \le j < h} TR(\sigma_j, \sigma_{j+1}) \rightarrow \\ &\qquad \exists \sigma'_0 \cdots \sigma'_\ell. \bigvee_i \left(\sigma'_0 = \curlyvee(\sigma_0) \wedge \bigwedge_{0 \le j < \ell'_i} TR(\sigma'_j, \sigma'_{j+1}) \ \wedge\ \sigma'_{\ell'_i} = \curlyvee(\sigma_{h'_i}) \right).\end{aligned} \tag{8}$$

For simplicity of the presentation, the preceding formula encodes only requirement (i) of Definition 6.3. Requirements (ii) and (iii) are straightforward to add.

Since (h_i, ℓ_i) are known to the analysis the number of such variables is finite. This formulation does, however, introduce quantifier alternation that did not exist in the original, deterministic setting. For a system that uses SMT to verify properties, this may have adverse effects.

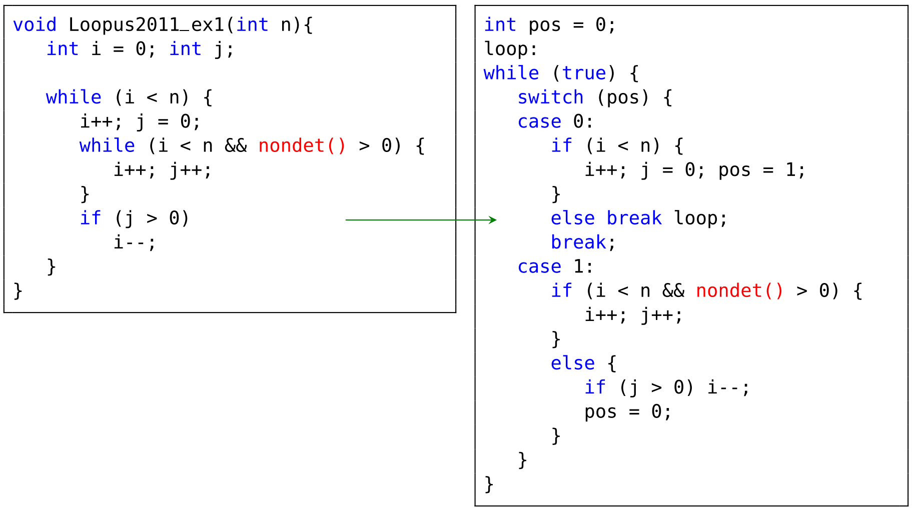
Bound on the number of occurrences of stuttering states that exceed $\widehat{k}$. We use the same approach to simplify the task of obtaining bounds b_i on the number of (h_i, ℓ_i)-stuttering states where $\frac{h_i}{\ell_i} > \widehat{k}$. Namely, we restrict such states σ to occur at most twice along each segment: at the beginning of the segment (meaning σ is a switch state) and at the end, totaling to at most $2d$ occurrences along each trace. The difference is that for non-deterministic programs, a state may have multiple outgoing traces and different stuttering behaviors in each of them. Therefore, to ensure that σ occurs at the end of the segments in all traces in which its stuttering behavior exceeds $\widehat{k}$, we require that whenever an outgoing trace τ of σ is (h_i, ℓ_i)-stuttering for $\frac{h_i}{\ell_i} > \widehat{k}$, the state at position h_i along τ is a last state. The encoding of this requirement is added to Equation (8).

Rank upper-bound property. This is, in fact, unchanged from our previous encoding. We had two cases in which rank bounds can be validated w.r.t. a given function $\hat{\curlyvee} : X \rightarrow X$: a single segment (Lemma 3.12) and rank preservation (Lemma 3.14). Both still hold when non-deterministic programs are considered.

Example 6.7. The program `Loopus2011_ex1` in Figure 11 demonstrates how a non-deterministic transition system may affect complexity analysis. The condition `nondet() > 0` may cause the inner loop to exit at any time, so it executes an arbitrary number of iterations between 0 and n. We show how our framework can handle reasoning about this loop. First, we transform the nested loop version to a single loop, shown to the right, using a standard transformation that introduces a program counter indicating whether the next iteration to execute is of the outer loop (`pos == 0`) or the inner loop (pos == 1).

Next, we detail the rank, squeezer, and simulation parameters as used in Theorem 3.10. We denote a state by $\sigma = \langle n, i, j, \texttt{pos} \rangle$. Then

- *Rank*: $\rho(\sigma) = n$
- *Squeezer*: $\curlyvee(\sigma) = \langle n - 1, i < n \ ?\ i : i - 1, j, \texttt{pos} \rangle$
- *Partition*: $d = 1$, $p_d(\sigma) = 1$
- *Stuttering pairs*: $\{(h_i, \ell_i)\}_i = \{(1, 1), (2, 2), (3, 0), (5, 1), (5, 3)\}$



```
void Loopus2011_ex1(int n){
   int i = 0; int j;

   while (i < n) {
      i++; j = 0;
      while (i < n && nondet() > 0) {
         i++; j++;
      }
      if (j > 0)
         i--;
   }
}
```

```
int pos = 0;
loop:
while (true) {
   switch (pos) {
   case 0:
      if (i < n) {
         i++; j = 0; pos = 1;
      }
      else break loop;
      break;
   case 1:
      if (i < n && nondet() > 0) {
         i++; j++;
      }
      else {
         if (j > 0) i--;
         pos = 0;
      }
   }
}
```

Fig. 11. An example program with a nested loop and a non-deterministic choice (from Loopus [34]). On the right, the same nested loop is rewritten as a flat loop.

- *Common stuttering ratio*: $\hat{k} = 1$
- *Base*: B contains all states of rank ≤ 1.

To demonstrate the simulation relation, we show that every reachable state is (h, ℓ)-stuttering for one of the stuttering pairs defined previously. We split into three cases depending on the value of i:

$i < n - 1$: Consider a state σ where $i < n - 1$, and let $\sigma' = \curlyvee(\sigma) = \langle n - 1, i, j, \texttt{pos}\rangle$. The only condition involving n in the program is `i < n`, and that condition is true both in σ and in σ'. As a result, if σ_1 is a successor of σ by executing command c, then executing c on σ' results in a state σ_1' with the same mutation of `i,j,pos`. Since $i < n - 1$, this implies that $\curlyvee(\sigma_1) = \sigma_1'$, as required for $(1, 1)$-stuttering.

$i = n$: The case of a state σ where $i = n$ is similar to the case where $i < n - 1$: both σ and $\sigma' = \curlyvee(\sigma) = \langle n - 1, i - 1, j, \texttt{pos}\rangle$ may execute the same command leading to σ_1 and $\sigma_1' = \curlyvee(\sigma_1)$, respectively. This holds since $i - 1 = n - 1$, and therefore the condition `i < n` is false in both σ and σ'. This is consistent with $(1, 1)$-stuttering. The only corner case occurs when $\texttt{pos} = 1$ and $j > 0$: both instances execute the command `i--; pos = 0`, leading to $\sigma_1 = \langle n, n - 1, j, 0\rangle$ and $\sigma_1' = \langle n - 1, n - 2, j, 0\rangle$. At this point, $\curlyvee(\sigma_1) = \langle n - 1, n - 1, j, 0\rangle \neq \sigma_1'$. One extra transition is required to synchronize the traces—observe the next step in each trace, $\sigma_2 = \langle n, n, j, 0\rangle$ and $\sigma_2' = \langle n - 1, n - 1, j, 0\rangle$. Now, $\curlyvee(\sigma_2) = \sigma_2'$, as needed, satisfying $(2, 2)$-stuttering.

$i = n - 1$: Consider the more subtle case of a state $\sigma = \langle n, n - 1, j, \texttt{pos}\rangle$. It is tricky because `i < n` is true in σ but false in $\sigma' = \curlyvee(\sigma) = \langle n - 1, n - 1, j, \texttt{pos}\rangle$. Fortunately, σ and σ' are very close to the end of their respective traces, allowing more forms of (h, ℓ)-stuttering. The choice of (h, ℓ) depends on the values of j and `pos`, as well as the non-determinisitic choice. A non-deterministic choice occurs when $\texttt{pos} = 1$; we use nd to denote the value chosen by `nondet()` in the current trace τ. The five cases are as follows:

(a) $\texttt{pos} = 0$;
(b) $\texttt{pos} = 1$, $nd > 0$ and $j = 0$;
(c) $\texttt{pos} = 1$, $nd > 0$ and $j > 0$;

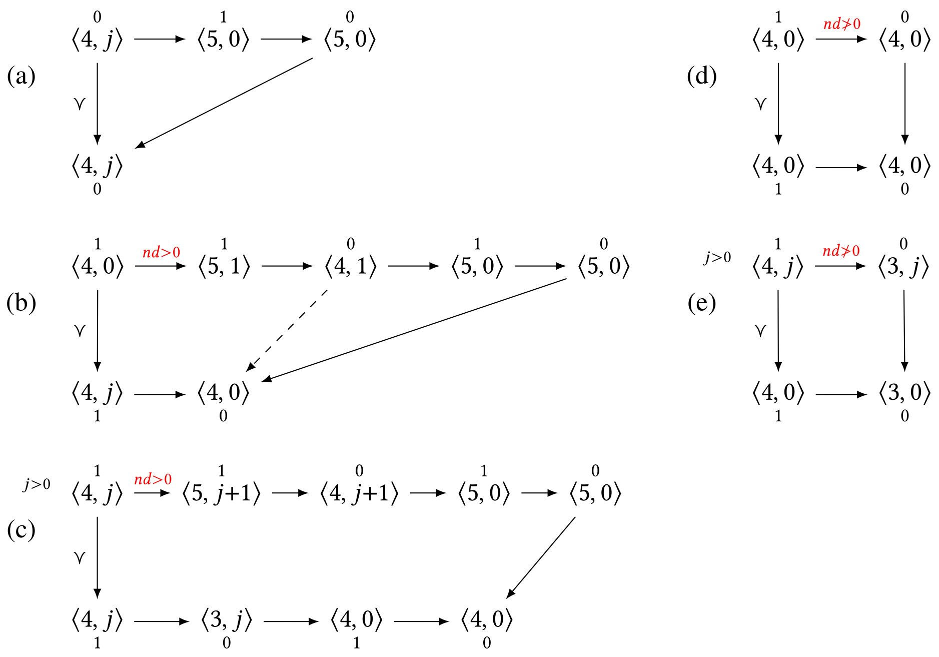


Fig. 12. The five cases of stuttering occurring in Section 6.7 when $\sigma = \langle n, n-1, j, \text{pos}\rangle$. For presentation purposes, these are shown for $n = 5$, but the treatment is general. The pairs $\langle i, j\rangle$ depict i and j in each state and the 0/1 values above/below display pos.

(d) $\text{pos} = 1$, $nd \not> 0$ and $j = 0$;
(e) $\text{pos} = 1$, $nd \not> 0$ and $j > 0$.

The five cases are depicted in Figure 12. For ease of reading, the figure shows for each state the values $\langle i, j\rangle$ with the value of pos on the outer rim. To make the diagram compact, we depict the property for $n = 5$ (and in the squeezed trace, $n = 4$), but the same result is obtained if $n, n-1, n-2$ are used in place of 5, 4, 3 in all states. As can be seen, the possible stuttering pairs are (3, 0), (5, 1), (5, 3), and (1, 1). The cases that are not (1, 1) end in a terminal state ($\text{pos} = 0 \wedge i = n$), as required by the definition.

With this, the recurrence relation obtained from 3.10 is

$$comp_x(n) = comp_x(n-1) + (3 + 4 + 2),$$

resulting in $O(n)$ complexity bound.

6.2 Pushing Non-determinism to the Inputs

An alternative approach to analyzing non-deterministic programs is via a reduction to deterministic programs with an extended input. Namely, given a non-deterministic program, we apply a rather standard transformation that introduces the following:

- A new array input for every non-deterministic choice instruction in the program, and
- An index variable for each array, initialized to 0.

Every non-deterministic instruction is replaced by an instruction that reads a value from the corresponding array and increments the index variable. The resulting program is deterministic and induces a deterministic transition system.

Let Σ be the set of states of the non-deterministic transition system that captures the semantics of the original program. Then the corresponding deterministic transition system is defined over an augmented set of states $\Sigma' = \Sigma \times \Delta$, where Δ is the auxiliary state storing the input arrays and their index variables. The initial states, $init'$, are augmented accordingly, where the auxiliary arrays may have an arbitrary content and the auxiliary index variables have value 0.

With this transformation, the complexity of an initial state $\sigma_0 \in init$ in the original non-deterministic transition system (per Definition 6.2) is equal to the maximal complexity of an initial state $\sigma_0' \in init'$ that augments σ_0 in the deterministic transition system (per Definition 3.2).

The analysis is applied to the deterministic transition system. Accordingly, all ingredients of the analysis are defined with respect to the augmented set of states Σ'. In particular, the squeezer is defined over the augmented states (i.e., it squeezes both the original state and the auxiliary state), thus implicitly matching non-deterministic choices of the higher-rank trace to non-deterministic choices of the lower-rank trace. The operation of the squeezer on the auxiliary state may be understood as a Skolem function that takes the place of the $\forall\exists$ quantifier alternation in the definition of partitioned simulation for non-deterministic transition systems (Equation (8)).

However, since we are interested in analyzing the worst-case complexity of the program over all non-deterministic choices (i.e., all values of the auxiliary input arrays), we do not allow the rank function $r : init' \to X$ to depend on the auxiliary state. Namely, we require that there exists a function $r_\Sigma : init \to X$ such that for $\sigma' \in init'$, $r(\sigma') = r_\Sigma(\sigma'_{|\Sigma})$, where $\sigma'_{|\Sigma}$ denotes the projection of σ' to Σ (i.e., removing the auxiliary state). Combined with Definition 3.4, this restriction on r ensures that $comp_x(\rho)$ provides an upper bound on the complexity of all initial states in $init'$ with rank ρ according to r, regardless of the non-deterministic choices, captured by the auxiliary state. As a result, $comp_x(\rho)$, which is the target of the analysis, also provides an upper bound on the complexity of all initial states in the non-deterministic transition system with rank ρ according to r_Σ.

Revisiting the example. In the program `Loopus2011_ex1`, the extended state is now $\sigma = \langle n, i, j, \text{pos}, \alpha, c_\alpha \rangle$. α is an auxiliary array of `int` values, and c_α is an index to α that is incremented on every call to `nondet()`. We now show that we can use the original definitions of Section 3 for the complexity analysis of the instrumented program.

All ingredients of Section 6.7 remain in place, except for $\curlyvee$, for which there is a trivial modification:

- *Squeezer*: $\curlyvee(\sigma) = \langle n-1, i < n \mathbin{?} i : i-1, j, \text{pos}, \alpha, c_\alpha \rangle$.

In other words, $\curlyvee$ has the same effect as before on the program variables and no effect at all on the auxiliary variables α, c_α.

The stuttering scenarios shown in Figure 12 are still valid in this case. The only difference is that instead of a non-deterministic choice transition, symbolized by the edges labeled $nd{>}0$, $nd{\not>}0$, all transitions are deterministic and the value of nd is read from the auxiliary state, $\alpha[c_\alpha]$. Consequently, the simulation property still holds and the same complexity bound applies.

6.3 Discussion

We have presented two approaches for extending the complexity analysis to non-deterministic programs. The approach presented in Section 6.1 follows the more traditional approach of modeling non-determinism via a transition relation; however, the resulting simulation requirement is more involved and, in particular, exhibits quantifier alternation (see Equation (8)), which is challenging for automated solvers. The alternative approach that models non-determinism using additional inputs, presented in Section 6.2, shifts this burden into the squeezer: the squeezer has to match the non-deterministic choices of the higher-rank and lower-rank traces, much like a Skolem function.

Both approaches are only able to extract recurrence relations for programs where the worst-case time complexity bound does not depend on the non-deterministic choices. This results from the restrictions imposed on the rank. In the first approach, the rank that is used in the recurrence relations is the rank of the initial state. Hence, it may not reflect non-deterministic choices performed during the execution. Similarly, when non-determinism is modeled via auxiliary state, the rank is not permitted to depend on the auxiliary state. For example, the time complexity of the following program depends on the (unbounded) sequence of non-deterministically chosen values of the variable y, and no bound may be expressed in terms of x. Hence, since the rank is restricted to be defined over x, no squeezer, rank, and rank bounding functions that satisfy all requirements exist in either approach.

For example, we have the following program:

```
unbounded(int x)
  while (x > 0) {
    x--;
    int y = nondet();
    while (y > 0) y--;
 }
```

The total running time, although finite, is unbounded. The number of iterations of the inner loop depends on the non-deterministically selected values for y. This can, in principle, be expressed as $\sum_{i=0}^{x} nd[i]$, where nd is the array of non-deterministic choices. However, the rank is purposefully barred from using this array, and, consequently, SqzComp will not find a valid rank, reporting the complexity bound ∞.

6.4 Evaluation

We evaluate our approach on non-deterministic programs using the encoding that pushes non-determinism to the inputs (Section 6.2). We use the benchmark suite of C^{4}B [13], which includes seven non-deterministic programs. A comparison of our results to C^{4}B, KoAT [11], Rank [6], Loopus [34], and SPEED [23] is presented in Table 2. The results reported for the external tools are taken from Carbonneaux et al. [13] since some of the tools do not take C programs as input, and for others, executables are not available.

7 RELATED WORK

This section focuses on exploring existing methods for *static* complexity analysis of *imperative* programs. Dynamic profiling and analysis [32] are a separate research area, more related to testing, and generally do not provide formal guarantees. We further focus on works that determine *asymptotic* complexity bounds and use the number of iterations executed as their cost model; we refrain from thoroughly covering previous techniques that analyze complexity at the instruction level.

Static cost analysis. The seminal work of Wegbreit [35] defined a two-step meta-framework where recurrence relations are extracted from the underlying program, then analyzed to provide closed-form upper bounds. Broadly speaking, cost relations are a generalized framework that captures the essence of most of the works mentioned in this section.

COSTA [4] and CoFloCo [21] infer cost relations of imperative programs written in Java and C, respectively. Cost relations resemble somewhat limited C procedures: they are capable of recursive calls to other cost relations, and they can handle non-determinism that arises either as

Table 2. Empirical Results of Running SqzComp on the Non-deterministic Programs from the C^{4}B Benchmarks Shown Alongside the Results Obtained from Previous Tools, Taken from Existing Publications

	Inferred Bound ($O(\cdot)$)						Time
Benchmark	KoAT	Rank	Loopus	SPEED	C^{4}B	SqzComp	SqzComp
speed_pldi10_ex1.c	—	—	—	n	n	2^n †	336 sec
speed_pldi10_ex3.c	—	n	n	n	n	n	<1 sec
speed_popl10_nested_multiple.c	—	$(m \dot{-} y) \cdot (n \dot{-} x)$	$(m \dot{-} y) + (n \dot{-} x)$	$(m \dot{-} y) + (n \dot{-} x)$	$(m \dot{-} y) + (n \dot{-} x)$	$(m \dot{-} y) + (n \dot{-} x)$††	9.6 sec
speed_popl10_nested_single.c	n	n	n	n	n	n	<1 sec
speed_popl10_sequential_single.c	n	n	n	n	n	n	<1 sec
speed_popl10_simple_single.c	n	n	n	n	n	n	<1 sec
t10.c	?	$x - y$	$x - y$	?	$x - y$	$x - y$	<1 sec

Note: The dash (—) means that the tool failed to produce a result. The question mark (?) means that the corresponding data entry is not available.
† A different squeezer exists that would allow discovering a bound of $O(n^2)$, but that squeezer is not in the space explored by our synthesis pass. SPEED and C^{4}B report a $O(n)$ bound because, in this particular benchmark, they are set up to count the number of times a specific line, marked with `tick`, executes; SqzComp lacks this feature.
†† Requires running SqzComp twice; the second run is to bound the base case of the first run. This is currently a manual process.

a consequence of direct `nondet()` in the program or as a result of inherent imprecision of static analysis. They define for every basic block of the program its own cost relation function, then form chains according to the control flow graph of the program. They use numerical abstract domains to support a context sensitive analysis of whether a chain of visits to specific basic blocks is feasible or not. Once all infeasible chains are removed, a specialized solving procedure that supports guarded equations determines an overall approximation of the heaviest chain, representing the max number of iterations.

SPEED [23] uses multiple counter instrumentations that are automatically inserted in various points in the code, initialized, and incremented. These ghost counters enable to infer an overall complexity bound by applying appropriate abstract interpretation handling numeric domains. In the work of Gulwani et al. [22] and Gulwani and Zuleger [24], code transformations are applied to represent multi-path loops and nested loops in a canonical way. Then, paths connecting pairs of "interesting" code points π_1, π_2 (loop headers, etc.) are identified, in a way that satisfies some properties. For instance, π_1 is reached twice *without* reaching π_2. The path property induces progress invariants, which are then analyzed to infer the overall complexity bound.

In the work of Lee et al. [30], an abstraction of the program to a *size-change graph* is defined, where transition edges of the control flow graph are annotated to capture sound overapproximation relations between integer variables. The graph is then searched for infinitely decreasing sequences, represented as words in an ω-regular language. This representation concisely characterizes program termination. The analysis of Zuleger et al. [37] then harnesses the size-change abstraction from Lee et al. [30] to analyze the complexity of imperative programs. First, they apply standard program transformations like path-wise analysis to summarize inner nested loops. Then, they heuristically define a set of scalar rank functions they call *norms*. These norms are somewhat similar to our rank function in the sense that they help to abstract away program parts that do not effect its complexity. The program is then represented as a size-change graph, and multi-path contextualization [31] prunes subsequent transitions that are infeasible.

In the work of Ben-Amram [9], *difference constraints* are introduced in the context of termination, to bound variables x' in current iteration with some y in previous iteration plus some constant c:

$x' \leq y + c$. Loopus [34] extends difference constraints to complexity analysis. Indeed, it is quite often the case that ideas from the area of program termination are assimilated in the context of complexity analysis and vice versa. They exploit the observation that typical operations on loop counters like increment, decrement, and resets are essentially expressible as difference constraints. They design an abstraction based on the domain of difference constraints and obtain relevant invariants which are then used in determining upper bounds. KoAT [11] is very similar, only it represents a program as an integer transition system and allows non-linear numerical constraints and ranking functions.

In the work of Winkler and Moser [36], an approach is presented for analyzing the runtime complexity of Logically Constrained Rewrite Systems (LCTRSs). They use the dependency graph, which describes the dependencies between rules in a rewrite system, to decompose the analysis, and apply several techniques to modularly extract expressions that bound variables that appear in the head of rules in terms of their initial values. Ultimately this results in recurrence relations that provide bounds on the worst-case time complexity of the rewrite system.

As we mentioned earlier, all of these approaches are based on identifying the progress of executions over time, characterizing the progress between two given points in the program. In contrast, our approach allows to reason over state size and compares whole executions.

Squeezers. The notion of squeezers was introduced in our earlier work [27] for the sake of safety verification. Similarly to the use of squeezers, Diffy [14, 15] uses difference relations to verify the safety of parameterized array manipulating programs by comparing an instance of the program to an instance with a smaller value of the parameter. As discussed in Section 1, the challenges in complexity analysis are different and require additional ingredients beyond squeezers (or difference relations). In other works [1, 2, 20], *well-structured transition systems* are introduced, where a **well-quasi order (wqo)** on the set of states induces a simulation relation. This property ensures decidability of safety verification of such systems (via a backward reachability algorithm). Our use of squeezers that decrease the rank of a state and induce a sort of a simulation relation may resemble the wqo of a well-structured transition system. However, there are several key differences: we do not require the order (which is defined on ranks) to be a wqo. Further, we do not require a simulation relation between any states whose ranks are ordered but only between a state and its squeezed counterpart. Notably, our work considers complexity analysis rather than safety verification.

8 CONCLUSION

This work introduces a novel framework for runtime complexity analysis. The framework supports derivation of recurrence relations based on inductive reasoning, where the form of induction depends on the choice of a squeezer (and rank bounding function). The new approach thus offers more flexibility than the classical methods where induction is coupled with the time dimension. For example, when the rank captures the "state size," the approach mimics induction over the space dimension, reasoning about whole traces, and alleviating the need to describe the intricate development of states over time. We demonstrate that such squeezers and rank bounding functions, which we manage to synthesize automatically, facilitate complexity analysis for programs that are beyond reach for existing methods. Thanks to the simplicity and compactness of these ingredients, even a rather naive enumeration was able to find them efficiently.

REFERENCES

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. 1996. General decidability theorems for infinite-state systems. In *Proceedings of the 1996 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, Los Alamitos, CA, 313–321.

[2] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. 2000. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.* 160, 1–2 (2000), 109–127.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Static Analysis*, María Alpuente and Germán Vidal (Eds.). Springer, Berlin, Germany, 221–237.

[4] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2007. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In *Proceedings of the 6th International Symposium on Formal Methods for Components and Objects (FMCO'07): Revised Lectures.* 113–132.

[5] Elvira Albert, Miquel Bofill, Cristina Borralleras, Enrique Martin-Martin, and Albert Rubio. 2019. Resource analysis driven by (conditional) termination proofs. *Theory Pract. Log. Program.* 19, 5–6 (2019), 722–739. https://doi.org/10.1017/S1471068419000152

[6] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proceedings of the International Static Analysis Symposium.* 117–133.

[7] Diego Esteban Alonso-Blas and Samir Genaim. 2012. On the limits of the classical approach to cost analysis. In *Static Analysis*, Antoine Miné and David Schmidt (Eds.). Springer, Berlin, Germany, 405–421.

[8] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, et al. 2015. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner (Eds.). NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25.

[9] Amir M. Ben-Amram. 2008. Size-change termination with difference constraints. *ACM Trans. Program. Lang. Syst.* 30, 3 (May 2008), Article 16, 31 pages.

[10] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas Reps. 2020. Templates and recurrences: Better together. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20).* ACM, New York, NY, 688–702.

[11] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating runtime and size complexity analysis of integer programs. In *Tools and Algorithms for the Construction and Analysis of Systems.* Lecture Notes in Computer Science, Vol. 8413. Springer, 140–155.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08).* 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[13] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 467–478.

[14] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. 2020. Verifying array manipulating programs with full-program induction. In *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'20).* 22–39.

[15] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. 2021. Diffy: Inductive reasoning of array programs using difference invariants. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV'21): Part II.* 911–935.

[16] Koen Claessen and John Hughes. 2011. QuickCheck: A lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Not.* 46, 4 (2011), 53–64.

[17] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'78).* ACM, New York, NY, 84–96.

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems.* Lecture Notes in Computer Science, Vol. 4963. Springer, 337–340.

[19] Saumya K. Debray and Nai-Wei Lin. 1993. Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst.* 15, 5 (Nov. 1993), 826–875.

[20] Alain Finkel and Philippe Schnoebelen. 1998. Well-structured transition systems everywhere! *Theor. Comput. Sci.* 256, 1 (1998), 2001.

[21] Antonio Flores-Montoya. 2016. Upper and lower amortized cost bounds of programs expressed as cost relations. In *FM 2016: Formal Methods.* Lecture Notes in Computer Science, Vol. 9995. Springer, 254–273.

[22] Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09).* ACM, New York, NY, 375–385.

[23] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: Precise and efficient static estimation of program computational complexity. *ACM SIGPLAN Not.* 44, 1 (2009), 127–139. I http://dblp.uni-trier.de/db/conf/popl/popl2009.html#GulwaniMC09.

[24] Sumit Gulwani and Florian Zuleger. 2010. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, NY, 292–304. https://doi.org/10.1145/1806596.1806630

[25] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource aware ML. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 7358. Springer, 781–786.

[26] Jan Hoffmann and Martin Hofmann. 2010. Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs (extended version). In *Proceedings of the 19th European Conference on Programming Languages and Systems*.

[27] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2020. Putting the squeeze on array programs: Loop verification via inductive rank reduction. In *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, Vol. 11990. Springer, 112–135.

[28] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2021. Run-time complexity bounds using squeezers. In *Programming Languages and Systems*. Lecture Notes in Computer Science, Vol. 12648. Springer, 320–347. https://doi.org/10.1007/978-3-030-72019-3_12

[29] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods Syst. Des.* 48, 3 (2016), 175–205. https://doi.org/10.1007/s10703-016-0249-4

[30] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. ACM, New York, NY, 81–92.

[31] Panagiotis Manolios and Daron Vroon. 2006. Termination analysis with calling context graphs. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer, Berlin, Germany, 401–414.

[32] Edison Mera, Pedro López-García, Germán Puebla, Manuel Carro, and Manuel V. Hermenegildo. 2007. Combining static analysis and profiling for estimating execution times. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*. 140–154.

[33] Rickard Nilsson. 2009. ScalaCheck: Property-Based Testing for Scala. Retrieved April 5, 2022 from https://www.scalacheck.org.

[34] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2017. Complexity and resource bound analysis of imperative programs using difference constraints. *J. Autom. Reasoning* 59, 1 (2017), 3–45.

[35] Ben Wegbreit. 1975. Mechanical program analysis. *Commun. ACM* 18, 9 (Sept. 1975), 528–539.

[36] Sarah Winkler and Georg Moser. 2020. Runtime complexity analysis of logically constrained rewriting. In *Proceedings of the 30th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'20)*. 37–55.

[37] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound analysis of imperative programs with the size-change abstraction. In *Static Analysis*, Eran Yahav (Ed.). Springer, Berlin, Germany, 280–297.

Received May 2021; revised November 2021; accepted January 2022