Quantifiers on Demand

Arie Gurfinkel[1], Sharon Shoham[2], and Yakir Vizel[3]

[1] University of Waterloo
[2] Tel Aviv University
[3] The Technion

Abstract. Automated program verification is a difficult problem. It is undecidable even for transition systems over Linear Integer Arithmetic (LIA). Extending the transition system with theory of Arrays, further complicates the problem by requiring inference and reasoning with universally quantified formulas. In this paper, we present a new algorithm, QUIC3, that extends IC3 to infer universally quantified invariants over the combined theory of LIA and Arrays. Unlike other approaches that use either IC3 or an SMT solver as a black box, QUIC3 carefully manages quantified generalization (to construct quantified invariants) and quantifier instantiation (to detect convergence in the presence of quantifiers). While QUIC3 is not guaranteed to converge, it is guaranteed to make progress by exploring longer and longer executions. We have implemented QUIC3 within the Constrained Horn Clause solver engine of Z3 and experimented with it by applying QUIC3 to verifying a variety of public benchmarks of array manipulating C programs.

1 Introduction

Algorithmic logic-based verification (ALV) is one of the most prominent approaches for automated verification of software. ALV approaches use SAT and SMT solvers to reason about bounded program executions; and generalization techniques, such as interpolation, to lift the reasoning to unbounded executions. In recent years, IC3 [8] (originally proposed for hardware model checking) and its extensions to Constrained Horn Clauses (CHC) over SMT theories [21,24] has emerged as the most dominant ALV technique. The efficiency of the IC3 framework is demonstrated by success of such verification tools as SEAHORN [19].

The IC3 framework has been successfully extended to deal with arithmetic [21], arithmetic and arrays [24], and universal quantifiers [23]. However, no extension supports the *combination* of all three. Extending IC3 to Linear Integer Arithmetic (LIA), Arrays, and Quantifiers is the subject of this paper. Namely, we present a technique to discover universally quantified solutions to CHC over the theories of LIA and Arrays. These solutions correspond to universally quantified inductive invariants of array manipulating programs.

For convenience of presentation, we present our approach over a transition system modelled using the theories of Linear Integer Arithmetic (LIA) and Arrays, and not the more general, but less intuitive, setting of CHCs. Inductive invariants of such transition systems are typically quantified, which introduces two major challenges: (i) quantifiers tremendously increase the search space for a candidate inductive invariant, and (ii) they require deciding satisfiability of quantified formulas – itself an undecidable problem.

Existing ALV techniques for inferring universally quantified arithmetic invariants either restrict the shape of the quantifiers and reduce to quantifier free inference [7,29,20], or guess quantified invariants from bounded executions [1].

In this paper, we introduce QUIC3 – an extension of IC3 [8,21,25] to universally quantified invariants. Rather than fixing the shape of the invariant, or discovering quantifiers as a post-processing phase, QUIC3 computes the necessary quantifiers *on demand* by taking quantifiers into account during the search for invariants. The key ideas are to allow existential quantifiers in proof obligations (or, counterexamples to induction) so that they are *blocked* by universally quantified lemmas, and to extend lemma generalization to add quantifiers.

Generating quantifiers on demand gives more control over the inductiveness checks. These checks (i.e., pushing in IC3) require deciding satisfiability of universally quantified formulas over the combined theory of Arrays and LIA. This is undecidable, and is typically addressed in SMT solvers by *quantifier instantiation* in which a universally quantified formula $\forall x \cdot \varphi(x)$ is approximated by a finite set of ground instances of φ. SMT solvers, such as Z3 [12], employ sophisticated heuristics (e.g., [15]) to find a sufficient set of instantiations. However, the heuristics are only complete in limited situations (recall, the problem is undecidable in general), and it is typical for the solver to return *unknown*, or, even worse, diverge in an infinite set of instantiations.

Instead of using an SMT solver as a black-box, QUIC3 generates and maintains a set of instantiations on demand. This ensures that QUIC3 always makes progress and is never stuck in a single inductiveness check. The generation of instances is driven by the *blocking* phase of IC3 and is supplemented by traditional pattern-based triggers. Generating both universally quantified lemmas and their instantiations on demand, driven by the property, offers additional flexibility compared to the eager quantifier instantiation approach of [7,29,20].

Combining the search for all of the ingredients (quantified and quantifier-free formulas, and instantiations) in a single procedure improves the control over the verification process. For example, even though there is no guarantee of convergence (the problem is, after all, undecidable), we guarantee that QUIC3 makes progress, exploring more of the program, and discovering a counter-example (even the shortest one) if it exists.

While our intended target is program verification, we have implemented QUIC3 in a more general setting of Constrained Horn Clauses (CHC). We build on the Generalized PDR engines [21,25] in Z3. The input is a set of CHC in SMT-LIB format, and the output is a universally quantified inductive invariant, or a counter-example. To evaluate QUIC3, we have used array manipulating C programs from SV-COMP. We show that our implementation is competitive and can automatically discover non-trivial quantified invariants.

In summary, the paper makes the following contributions: (a) extends IC3 framework to support quantifiers; (b) develops quantifier generalization techniques; (c) develops techniques for discovering quantifier instantiations during verification; and (d) reports on our implementation for software verification.

2 Preliminaries

Logic. We consider First Order Logic modulo the combined theory of Linear Integer Arithmetic (LIA) and Arrays. We denote the theory by $\mathcal{T}$ and the logic by $FOL(\mathcal{T})$. We assume that the reader is familiar with the basic notions of $FOL(\mathcal{T})$ and provide only a brief description to set the notation. Formulas in $FOL(\mathcal{T})$ are defined over a signature Σ which includes sorts int and array, where sort int is also used as the sort of the array indices and data. We assume that the signature Σ includes equality (=), interpreted functions, predicates, and constants of arithmetic (i.e., the functions $+$, $-$, $*$, the predicates $<$, $\leq$, and the constants 1, 2, etc.) and of arrays (i.e., the functions sel and $store$).

In addition, Σ may be extended with uninterpreted constants. In particular, we assume that Σ includes special *Skolem* uninterpreted constants $SK = \{sk_i\}$ of sort int for i in natural numbers.

We denote by $\Sigma_{\mathcal{T}}$ the interpreted part of Σ, and by $X \subseteq \Sigma$ the set of uninterpreted constants (e.g., a or sk_i, but not 1). In the sequel we write $\varphi(X)$, and say that φ is defined over X, to denote that φ is defined over signature $\Sigma = \Sigma_{\mathcal{T}} \cup X$. We write $Const(\varphi) \subseteq X$ for the set of all uninterpreted constants that appear in φ. In the rest of the paper, whenever we refer to constants, we only refer to the uninterpreted ones.

We write T for the set of terms of $FOL(\mathcal{T})$, and V for the set of (sorted) variables. We assume that int variables in V are of the form v_i, where i is a natural number. Thus, we can refer to all such variables by their numeric name. For a formula φ, we write $Terms(\varphi) \subseteq T$ and $FVars(\varphi) \subseteq V$ for the terms and free variables of φ, respectively.

A substitution $\sigma : V \to T$ is a partial mapping from V to terms in T that pertains to the sort constraints. We write $dom(\sigma)$ to denote the domain of σ, and $range(\sigma)$ to denote its range. For a formula φ, we write $\varphi\sigma$ for the result of applying substitution σ to φ. Abusing notation, we write $\emptyset$ for an empty substitution, i.e., a substitution σ such that $dom(\sigma) = \emptyset$. Given two substitutions σ_1 and σ_2, we write $(\sigma_1 \mid \sigma_2)$ for a composition of substitutions defined such that: $(\sigma_1 \mid \sigma_2)(x) = \sigma_1(x)$ if $x \in dom(\sigma_1)$, and $\sigma_2(x)$, otherwise. We define a special *Skolem substitution* $sk : V \to T$ such that $sk(v_i) = sk_i$ for $sk_i \in SK$. Given a formula L, we write L_{sk} for Lsk, and given a substitution σ.

We write $abs(U, \varphi) = (\psi, \sigma)$ for an abstraction function that given a set of uninterpreted constants U and a formula φ returns an abstraction ψ of φ in which the constants are replaced by free variables, as well as a substitution σ that records the mapping of variables back to the constants that they abstract. Formally, we require that $abs(U, \varphi) = (\psi, \sigma)$ satisfies the following: $\psi\sigma = \varphi$, $dom(\sigma) = FVars(\psi) \setminus FVars(\varphi)$, and $U \cap Terms(\psi) = \emptyset$. The requirements ensure that abs abstracts all uninterpreted constants in U, and σ maps the newly introduced variables back to the constants. Furthermore, we require that for every skolem constant sk_i in U, $abs(U, \varphi)$ abstracts sk_i in φ to v_i in ψ, and accordingly, $\sigma(v_i) = sk_i$. This ensures that applying skolemization, followed by abstraction of SK, reintroduces the same variables and does not result in variable renaming. That is, $abs(SK, \varphi_{sk}) = (\varphi, \_)$.

We write $\forall\varphi$ for a formula obtained from φ by universally quantifying all free variables of φ, and $\exists\varphi$ for a formula obtained by existential quantification, respectively. For convenience, given a set of constants U and a ground formula φ (i.e., a formula where all terms are ground), we write $\exists U \cdot \varphi$ for $\exists \psi$, where $(\psi, \sigma) = abs(U, \varphi)$. We write $\varphi \Rightarrow \psi$ do denote the validity of $\varphi \rightarrow \psi$.

Model Based Projection. Given a ground formula φ, a model M of φ, and a set of uninterpreted constants $U \subseteq Const(\varphi)$, (partial, or incomplete) Model Based Projection, MBP, is a function $\textsc{pMbp}(U, \varphi, M) = (\psi, W)$ such that 1. ψ is a ground monomial (i.e., conjunction of ground literals), 2. $W \subseteq U$ and $Const(\psi) \subseteq Const(\varphi) \setminus (U \setminus W)$, 3. $\psi \Rightarrow (\exists U \setminus W \cdot \varphi)$, 4. $M \models \psi$, 5. $\textsc{pMbp}$ is finite ranging in its third argument: for a fixed U and φ, the set $\{\textsc{pMbp}(U, \varphi, M) \mid M \models \varphi\}$ is finite. Intuitively, the monomial ψ underapproximates (implies) the result of eliminating the existential quantifiers pertaining to $U \setminus W$ from φ (where quantifier elimination itself may not even be defined). It, therefore, represents one of the ways of satisfying the result of quantifier elimination. The underapproximation ψ is chosen such that it is consistent with the provided model M. In this paper, MBP is used as a way to underapproximate the pre-image of a set of states represented implicitly by some formula.

An MBP is called *complete* if W is always empty. A complete MBP for Linear Arithmetic has been presented in [25] and a partial MBP for the theory of arrays has been presented in [24]. Importantly, in the partial MBP of [24], the remaining set of constants, W, never contains any constant of sort array. We refer the readers to [25,24] and to [6] for details. A complete MBP under-approximates quantifier elimination relative to a given model. Such an MBP can only exist if the underlying theory admits quantifier elimination. Since the theory of arrays does not admit quantifier elimination it only admits a partial MBP.

In the paper, we further require an MBP to eliminate all the constants of sort array from U, such as the MBP of [24].

Interpolation. Given a ground formula A, and a ground monomial B such that $A \Rightarrow \neg B$, (partial) interpolation, ITP, is a function $\textsc{pItp}(A, B) = (\varphi, U)$, s.t. 1. φ is a ground clause (i.e., a disjunction of ground literals), 2. $U \subseteq Const(B) \setminus Const(A)$ and $Const(\varphi) \subseteq (Const(A) \cap Const(B)) \cup U$, 3. $A \Rightarrow \forall U \cdot \varphi$, and 4. $\varphi \Rightarrow \neg B$. The set of constants U denotes the constants of φ that exceed the set of shared constants of A and B. An interpolation procedure is complete if for any pair A, B, the returned set U is always empty. The formula φ produced by a complete interpolation procedure is called an *interpolant* of A and B. Note that our definitions admit a trivial partial interpolation procedure defined as $\textsc{pItp}_{triv}(A, B) = (\neg B, Const(B) \setminus Const(A))$.

Safety problem. We represent transition systems via formulas in $FOL(\mathcal{T})$. The states of the system correspond to structures over a signature $\Sigma = \Sigma_{\mathcal{T}} \cup X$, where X denotes the set of (uninterpreted) constants. The constants in X are used to represent program variables. A *transition system* is a pair $\langle Init(X), Tr(X, X') \rangle$, where $Init$ and Tr are quantifier-free ground formulas in $FOL(\mathcal{T})$. $Init$ represents the initial states of the system and Tr represents the transition relation. We write $Tr(X, X')$ to denote that Tr is defined over the signature $\Sigma_{\mathcal{T}} \cup X \cup X'$, where X is used to represent the pre-state of a transition, and $X' = \{a' \mid a \in X\}$ is used to represent the post-state. A *safety problem* is a triple $\langle Init(X), Tr(X, X'), Bad(X) \rangle$, where $\langle Init, Tr \rangle$ is a transition system and Bad is a quantifier-free ground formula in $FOL(\mathcal{T})$ representing a set of bad states.

The safety problem $\langle Init(X), Tr(X, X'), Bad(X)\rangle$ has a *counterexample of length* k if the following formula is satisfiable:

$$BMC_k(Init, Tr, Bad) = Init(X_0) \wedge \bigwedge_{i=0}^{k-1} Tr(X_i, X_{i+1}) \wedge Bad(X_k),$$

where $X_i = \{a_i \mid a \in X\}$ is a copy of the constants used to represent the state of the system after the execution of i steps. The transition system is *safe* if the safety problem has no counterexample, of any length.

Interpolation sequence and inductive invariants. An *interpolation sequence of length* k for a safety problem $\langle Init(X), Tr(X, X'), Bad(X)\rangle$ is a sequence of formulas $I_1(X), \dots, I_k(X)$ such that (i) $Init(X) \Rightarrow I_1(X)$, (ii) $I_j(X) \wedge Tr(X, X') \Rightarrow I_{j+1}(X')$ for every $1 \leq j \leq k-1$, and (iii) $I_k(X) \Rightarrow \neg Bad(X)$. If an interpolation sequence of length k exists, then the transition system has no counterexample of length k. An *inductive invariant* is a formula $Inv(X)$ such that (i) $Init(X) \Rightarrow Inv(X)$, (ii) $Inv(X) \wedge Tr(X, X') \Rightarrow Inv(X')$, and (iii) $Inv(X) \Rightarrow \neg Bad(X)$. If such an inductive invariant exists, then the transition system is safe.

3 Quantified IC3

In this section, we present QUIC3 – a procedure for determining a safety of a transition system by inferring quantified inductive invariants. Given a safety problem, QUIC3 attempts to discover an inductive invariant $Inv(X)$ as a universally-quantified formula of $FOL(\mathcal{T})$ (where quantification is restricted to variables of sort int) or produce a counterexample.

We first present QUIC3 as a set of rules, following the presentation style of [21,25,5,24,18]. We focus on the data structures, the key differences between QUIC3 and IC3, and soundness of the rules. An imperative procedure based on these rules is presented in Section 4. We assume that the reader is familiar with the basics of IC3. Throughout the section, we fix a safety problem $P = \langle Init(X), Tr(X, X'), Bad(X)\rangle$, and assume that $Init$, Tr and Bad are quantifier free ground formulas. For convenience of presentation, we use the notation $\mathcal{F}(A)$ to denote the formula $(A(X) \wedge Tr(X, X')) \vee Init(X')$ that corresponds to the forward image of A over the Tr extended by the initial states.

The rules of QUIC3 are shown in Alg. 1. Similar to IC3, QUIC3 maintains a queue $\mathcal{Q}$ of proof obligations (POBs), and a monotone inductive trace $\mathcal{T}$ of frames containing lemmas at different levels. However, both the proof obligations and the lemmas maintained by QUIC3 are quantified.

Quantified Proof Obligations. Each POB in $\mathcal{Q}$ is a triple $\langle m, \sigma, i\rangle$, where m is a monomial over X such that $FVars(m)$ are of sort int, σ is a substitution such that $FVars(m) \subseteq dom(\sigma)$ and $range(\sigma) \subseteq X' \cup SK$, and i is a natural number representing the frame index at which the POB should be either blocked or extended. The POB $\langle m, \sigma, i\rangle$ expresses an obligation to show that no state satisfying $\exists m$ is reachable in i steps of Tr. The substitution σ records the specific instance of the free variables in frame $i+1$ that were abstracted during construction of m. Whenever the POB is blocked, a universally

Input: A safety problem $\langle Init(X), Tr(X, X'), Bad(X)\rangle$.
Assumptions: $Init$, Tr and Bad are quantifier free.
Data: A POB queue $\mathcal{Q}$, where a POB $c \in \mathcal{Q}$ is a triple $\langle m, \sigma, i\rangle$, m is a conjunction of literals over X and free variables, σ is a substitution s.t. $m\sigma$ is ground, and $i \in \mathbb{N}$. A level N. A quantified trace $\mathcal{T} = Q_0, Q_1, \ldots$, where for every pair $(\ell, \sigma) \in Q_i$, ℓ is a quantifier-free formula over X and free variables and σ a substitution s.t. $\ell\sigma$ is ground.
Notation: $\mathcal{F}(A) = (A(X) \wedge Tr(X, X')) \vee Init(X')$; $qi(Q) = \{\ell\sigma \mid (\ell, \sigma) \in Q\}$; $\forall Q = \{\forall \ell \mid (\ell, \sigma) \in Q\}$.

Output: *Safe* or *Cex*
Initially: $\mathcal{Q} = \emptyset$, $N = 0$, $Q_0 = \{(Init, \emptyset)\}$, $\forall i > 0 \cdot Q_i = \emptyset$.
repeat

- **Safe** If there is an $i < N$ s.t. $\forall Q_i \subseteq \forall Q_{i+1}$ **return** *Safe*.
- **Cex** If there is an m, σ s.t. $\langle m, \sigma, 0\rangle \in \mathcal{Q}$ **return** *Cex*.
- **Unfold** If $qi(Q_N) \Rightarrow \neg Bad$, then set $N \leftarrow N + 1$.
- **Candidate** If for some m, $m \Rightarrow qi(Q_N) \wedge Bad$, then add $\langle m, \emptyset, N\rangle$ to $\mathcal{Q}$.
- **Predecessor** If $\langle m, \xi, i+1\rangle \in \mathcal{Q}$ and there is a model M s.t. $M \models qi(Q_i) \wedge Tr \wedge (m'_{sk})$, add $\langle \psi, \sigma, i\rangle$ to $\mathcal{Q}$, where $(\psi, \sigma) = abs(U, \varphi)$ and $(\varphi, U) = \text{PMBP}(X' \cup SK, Tr \wedge m'_{sk}, M)$.
- **NewLemma** For $0 \leq i < N$, given a POB $\langle m, \sigma, i+1\rangle \in \mathcal{Q}$ s.t. $\mathcal{F}(qi(Q_i)) \wedge m'_{sk}$ is unsatisfiable, and $L' = \text{ITP}(\mathcal{F}(qi(Q_i)), m'_{sk})$, add (ℓ, σ) to Q_j for $j \leq i+1$, where $(\ell, \_) = abs(SK, L)$.
- **Push** For $0 \leq i < N$ and $((\varphi \vee \psi), \sigma) \in Q_i$, if $(\varphi, \sigma) \notin Q_{i+1}$, $Init \Rightarrow \forall \varphi$ and $(\forall \varphi) \wedge \forall Q_i \wedge qi(Q_i) \wedge Tr \Rightarrow \forall \varphi'$, then add (φ, σ) to Q_j, for all $j \leq i+1$.

until ∞;

Algorithm 1: The rules of QUIC3 procedure.

quantified lemma $\forall \ell$ is generated in frame i (as a generalization of $\forall \neg m$), and, σ is used to discover the specific instance of $\forall \ell$ that is necessary to prevent generating the same POB again.

Quantified Inductive Trace. A quantified monotone inductive trace $\mathcal{T}$ is a sequence of sets Q_i. Each Q_i is a set of pairs, where for each pair (ℓ, σ) in Q_i, ℓ is a formula over X, possibly with free variables, such that all free variables $FVars(\ell)$ are of sort int, and σ is a substitution such that $FVars(\ell) \subseteq dom(\sigma)$ and $range(\sigma) \subseteq X' \cup SK$. Intuitively, a pair (ℓ, σ) corresponds to a universally quantified lemma $\forall \ell$ and its ground instance $\ell\sigma$. If ℓ has no free variables, it represents a ground lemma (as in the original IC3). We write $\forall Q_i = \{\forall L \mid (L, \sigma) \in Q_i\}$ for the set of all ground and quantified lemmas in Q_i, and $qi(Q_i) = \{\ell\sigma \mid (\ell, \sigma) \in Q_i\}$ for the set of all instances in Q_i.

QUIC3 maintains that the trace $\mathcal{T}$ is inductive and monotone. That is, it satisfies the following conditions, where N is the size of $\mathcal{T}$:

$$Init \Rightarrow \forall Q_0 \qquad \forall 0 \leq i < N \cdot \forall Q_i \wedge Tr \Rightarrow \forall Q_{i+1} \qquad \forall Q_{i+1} \subseteq \forall Q_i$$

The first two conditions ensure inductiveness and the last ensures syntactic monotonicity. Both are similar to the corresponding conditions in IC3.

The rules. The rules **Safe**, **Cex**, **Unfold**, **Candidate** are essentially the same as their IC3 counterparts. The only exception is that, whenever the lemmas of frame i are required, the instances $qi(Q_i)$ of the quantified lemmas in Q_i are used (instead of $\forall Q_i$). This ensures that the corresponding satisfiability checks are decidable and do not diverge.

***Predecessor** rule.* **Predecessor** extends a POB $\langle m, \xi, i+1\rangle \in \mathcal{Q}$ from frame $i+1$ with a predecessor POB $\langle \psi, \sigma, i\rangle$ at frame i. The precondition to the rule is satisfiability of $qi(Q_i) \wedge Tr \wedge (m'_{sk})$. Note that all free variables in the current POB m are skolemized via the substitution sk (recall that all the free variables are of sort int) and all constants are primed.

Predecessor rule extends the corresponding rule of IC3 in two ways. First, POBs are generated using partial MBP. The $\text{pMBP}(X' \cup SK, Tr \wedge m'_{sk}, M)$ is used to construct a ground monomial φ over $X \cup X' \cup SK$, describing a predecessor of m'_{sk}. Whenever φ contains constants from $X' \cup SK$, these are abstracted by fresh free variables to construct a POB ψ over X. Thus, the newly constructed POB is not ground and its free variables are implicitly existentially quantified. (Since pMBP is guaranteed to eliminate all constants of sort array, the free variables are all of sort int). Second, the **Predecessor** maintains with the POB ψ the substitution σ that corresponds to the inverse of the abstraction used to construct ψ from φ, i.e., $\psi\sigma = \varphi$. It is used to introduce a ground instance that blocks ψ as a predecessor of $\langle m, \xi, i+1\rangle$ when the POB is blocked (see **NewLemma**).

The soundness of **Predecessor** (in the sense that it does not introduce spurious counterexamples) rests on the fact that every state in the generated POB has a Tr successor in the original POB. This is formalized as follows:

Lemma 1. *Let $\langle m, \xi, i+1\rangle \in \mathcal{Q}$ and let (ψ, σ, i) be the POB computed by **Predecessor**. Then, $(\exists \psi) \Rightarrow \exists X' \cdot (Tr \wedge \exists m')$.*

Proof. From the definition of **Predecessor**, $(\psi, \sigma) = abs(U, \varphi)$, where $(\varphi, U) = \text{pMBP}(X' \cup SK, Tr \wedge m'_{sk}, M)$. The set $U \subseteq X' \cup SK$ are the constants that were not eliminated by MBP. Then, by properties of pMBP, $\psi\sigma \Rightarrow \exists (X', SK) \setminus U \cdot Tr \wedge m'_{sk}$. Note that $(\exists U \cdot \varphi) = \exists \psi$. By abstracting U in φ and existentially quantifying over the resulting variables in both sides of the implication, we get that $\exists \psi \Rightarrow \exists X', SK \cdot Tr \wedge m'_{sk}$. Since SK does not appear in Tr, the existential quantification distributes over Tr: $\exists X', SK \cdot Tr \wedge m'_{sk} \equiv \exists X' \cdot (Tr \wedge \exists m')$. □

By induction and Lemma 1, we get that if $\langle \psi, \sigma, i\rangle$ is a POB in $\mathcal{Q}$, then every state satisfying $\exists \psi$ can reach a state in Bad.

***NewLemma** rule.* **NewLemma** creates a potentially quantified lemma ℓ and a corresponding instance $\ell\sigma$ to block a quantified POB $\langle m, \sigma, i+1\rangle$ at level $i+1$. Note that if ℓ is quantified, then while the instance $\ell\sigma$ is guaranteed to be new at level $i+1$, the lemma ℓ might already appear in Q_{i+1}. The lemma ℓ is first computed as in IC3, but using a skolemized version of the POB. Second, if any skolem constants remain in the lemma, then they are re-abstracted into the original variables. The corresponding instance of ℓ is determined by the substitution σ of the POB. Note that the instance $\ell\sigma$ is well defined since abs abstracts skolem constants back into the variables (of sort int) that introduced them, ensuring that $FVars(\ell) \subseteq dom(\sigma)$. Note further that if ℓ has no

```
void init_arrray(int[] A, int sz) {
  1: for (int i = 0; i < sz; i++)  A[i] = 0;
  2: j = nd(); assume(0 <= j && j < sz);
  3: assert(A[j] == 0);}
```

Fig. 1. An array manipulating program.

free variables, then the substitution σ is redundant and could be replaced by an empty substitution. (In fact, it is always sufficient to project σ to $FVars(\ell)$.)

The soundness of **NewLemma** follows form the fact that every lemma (ℓ, σ) that is added to the trace $\mathcal{T}$ keeps the trace inductive. Formally:

Lemma 2. *Let (ℓ, σ) be a quantified lemma added to Q_{i+1} by **NewLemma**. Then, $\mathcal{F}(\forall Q_i) \Rightarrow (\forall \ell')$.*

Proof. ℓ is $abs(SK, L)$, where $L' = \textsc{Itp}(\mathcal{F}(qi(Q_i)), m'_{sk})$. Therefore, $\mathcal{F}(qi(Q_i)) \wedge \neg L'$ is unsatisfiable. Let Ψ be $\mathcal{F}(\forall Q_i) \wedge (\neg \forall \ell')$, and assume, to the contrary, that Ψ is satisfiable. Since no constants from SK appear in $\mathcal{F}(\forall Q_i)$ and ℓ is $abs(SK, L)$, Ψ is equi-satisfiable to $\mathcal{F}(\forall Q_i) \wedge (\neg L')$. Let M be the corresponding model. Then, in contradiction, $M \models \mathcal{F}(qi(Q_i)) \wedge (\neg L')$. □

Rules **Predecessor** and **NewLemma** use m'_{sk} that is skolemized with our special skolem substitution where $sk(v_i) = sk_i$. We note that while the skolem constants in m'_{sk} are always a subset of SK and do not overlap with $X \cup X'$, they may overlap the existing skolem constants that appear in the rest of the formula (e.g., if the rest of the formula contains $qi(Q_{i-1})$, where the ground instances result from previously blocked POBs and, therefore, also contain skolem constants). In this sense, our skolemization appears non-standard. However, all the claims in this section only rely on the fact that the range of sk is SK and that SK is disjoint from $X \cup X'$, which holds for sk.

***Push** rule.* **Push** is similar to its IC3 counterpart. It propagates a (potentially quantified) lemma to the next frame. The key difference is the use of quantified formulas $\forall Q_i$ (and their instantiations $qi(Q_i)$ in the pre-condition of the rule. Thus, checking applicability of **Push** requires deciding validity of a quantified FOL formula, which is undecidable in general. In practice,, we use a weaker, but decidable, variant of these rules. In particular, we use a finite instantiation strategy to instantiate $\forall Q_i$ in combination with all of the instantiations $qi(Q_i)$ discovered by QUIC3 before theses rules are applied. This ensures progress (i.e., QUIC3 never gets stuck in an application of a rule) at an expense of completeness (some lemmas are not pushed as far as possible, which impedes divergence).

We illustrate the rules on a simple array-manipulating program `init_array` shown in Fig. 1. In the program, `assume` and `assert` stand for the usual assume and assert statements, respectively, and `nd` returns a non-deterministic value. We assume that the program is converted into a safety problem as usual. In this problem, a special variable pc is used to indicate the program counter. The first POB found by **Candidate** is $pc = 3 \wedge \mathsf{sel}(A, j) \neq 0$. Its predecessor, is $pc = 2 \wedge \mathsf{sel}(A, v_0) \neq 0 \wedge 0 \leq v_0 < sz$ and the corresponding substitution is $(v_0 \mapsto j)$. Note that since PMBP could not eliminate j, it was replaced by a free variable. Eventually, this POB is blocked, the lemma that is added is $\forall((pc = 2 \wedge 0 \leq v_0 < sz) \Rightarrow \mathsf{sel}(A, v_0) = 0)$.

```
N ← 0; Q_0 = {(Init, ∅)}
if Init ∧ Bad then
    return CEX
while (true) do
    N ← N + 1;    Q_N ← ∅
    if Quic3_MakeSafe(Bad, ∅, N) = CEX then
        return CEX
    if Quic3_Push() = SAFE then
        return SAFE
end
```

Fig. 2. Main Procedure (`Quic3_Main`). Wlog, we assume that Bad is a monomial.

Soundness. We conclude this section by showing that applying QUIC3 rules from Alg. 1 in any order is sound:

Lemma 3. *If* QUIC3 *returns* Cex*, then P is not safe (and there exists a counterexample). Otherwise, if* QUIC3 *returns* Safe*, then P is safe.*

Proof. The first case follows immediately from Lemma 1. The second case follows from the properties of the inductive trace maintained by QUIC3 that ensure that whenever *Safe* is returned (by **Safe** rule), a safe inductive invariant is obtained. Lemma 2 ensures that these properties are preserved whenever a new quantified lemma is added. Soundness of all other rules follows the same argument as the corresponding rules of IC3. □

In fact, QUIC3 ensures a stronger soundness guarantee:

Lemma 4. *In every step of* QUIC3*, for every $k < N$, the sequence $\{\forall Q_i\}_{i=1}^{k}$ is an interpolation sequence of length k for P.*

Thus, if QUIC3 reaches $N > k$, then there are no counterexample of length k.

4 Progress and Counterexamples

Safety verification of transition systems described in the theory of LIA and Arrays is undecidable in general. Thus, there is no expectation that QUIC3 always terminates. None-the-less, it is desirable for such a procedure to have strong progress guarantees – the longer it runs, the more executions are explored. In this section, we show how to orchestrate the rules defining QUIC3 (shown in Alg. 1) into an effective procedure that guarantees progress in exploration and produces a shortest counterexample, if it exists.

Realization of QUIC3. Fig. 2 depicts procedure `Quic3_Main` – an instance of QUIC3 where each iteration, starting from $N = 0$, consists of a `Quic3_MakeSafe` phase followed by a `Quic3_Push` phase. The `Quic3_MakeSafe` phase, described in Fig. 3, starts by initializing $\mathcal{Q}$ to the POB $(Bad, \emptyset, N)$ (this is a degenerate application of **Candidate** that is sufficient when Bad is a monomial). It then applies **Predecessor** and **NewLemma** iteratively until either a counterexample is found or $\mathcal{Q}$ is emptied. **NewLemma** is preceded by an optional generalization procedure (Line 16) that may introduce additional quantified variables and record the constants that they originated

Input: (Cube m_0, Substitution σ_0, Level i_0)
Data: Queue $\mathcal{Q}$ of triples $\langle m, \sigma, i\rangle$, where m is a cube, σ is a substitution and i is a level

```
Q = ∅
// Apply Candidate rule
Add(Q, ⟨m_0, σ_0, i_0⟩)
while ¬Empty(Q) do
    ⟨m, ξ, i⟩ ← Top(Q)
    if i = 0 then
        // Apply Cex rule; Found a counterexample
        return CEX
    M ← SAT(qi(Q_{i−1}) ∧ Tr ∧ (m'_sk))
    if M ≠ ⊥ then
        // Apply Predecessor rule
        (φ, U) ← PMBP(X' ∪ SK, Tr ∧ m'_sk, M)
        (ψ, σ) ← abs(U, φ)
        Add(Q, ⟨ψ, σ, i − 1⟩)
    else
        Remove(Q, ⟨m, ξ, i⟩)
        L' ← ITP(qi(Q_{i−1} ∧ Tr), m'_sk)
        // Abstract all skolem constants
        (ℓ, _) ← abs(SK, L)
        // Optional quantified generalization (see Sec. 5)
        (ℓ, ξ) ← QGen(ℓ, ⟨m, ξ, i⟩)
        // Apply NewLemma rule
        forall j ≤ i, Q_j ← Q_j ∪ {(ℓ, ξ)}
end
return BLOCKED
```

Fig. 3. `Quic3_MakeSafe` procedure of QUIC3.

from by extending the substitution ξ. We defer discussion of this procedure to Section 5; in the simplest case, it will return the same lemma with the same substitution ξ. At the end of `Quic3_MakeSafe`, the trace $(Q_i)_i$ is an interpolation sequence of length N. The `Quic3_Push` applies **Push** iteratively from frame $i = 1$ to $i = N$. The corresponding satisfiability queries are restricted to use the existing instances of quantified lemmas and a finite set of instantiations pre-determined by heuristically chosen triggers. If, as a result of pushing, two consecutive frames become equal (rule **Safe**), `Quic3_Main` returns *Safe*.

Progress. Recall that we use a *deterministic* skolemization procedure. Namely, for a POB $\langle m, \xi, i\rangle$, in every satisfiability check of the form $qi(Q_{i-1}) \wedge Tr \wedge (m'_{sk})$, the same skolem substitution (defined by $sk(v_i) = sk_i$) is used in m'_{sk}, even if the rest of the formula (i.e., $qi(Q_{i-1})$) changes. The benefit of using a deterministic skolemization procedure is that it ensures that all applications of PMBP in **Predecessor** use exactly the same formula $Tr \wedge m'_{sk}$ and exactly the same set of constants. As a result, the number of predecessors (POBs) generated by applications of **Predecessor** for each POB is bounded by the finite range of PMBP in its third (model) argument:

Lemma 5. *If a deterministic skolemization is used, then for each POB $\langle m, \xi, i\rangle$, the number of POBs generated by applying **Predecessor** on $\langle m, \xi, i\rangle$ is finite.*

Proof. For simplicity, we ignore the application of quantified generalization; the proof extends to handle it as well. After a quantified lemma (ℓ, ξ) is added to Q_{i-1}, every model $M \models qi(Q_{i-1}) \wedge Tr \wedge m'_{sk}$ that is discovered when applying **Predecessor** on $\langle m, \xi, i\rangle$ will be such that $M \models \ell\xi$. Recall that the lemma was generated by a POB $\langle \varphi, \sigma, i-1\rangle$ that was blocked since $qi(Q_{i-2}) \wedge Tr \wedge \varphi'_{sk}$ was unsatisfiable, and $(\ell, \_) = abs(SK, L)$ where $L' = \textsc{Itp}(qi(Q_{i-2} \wedge Tr), \varphi'_{sk})$. Therefore $L \wedge \varphi_{sk} \equiv \bot$. Since abs maps each skolem constant back to the variable that introduced it, we have that the skolems in L are abstracted to the original variables from φ. Hence, $\ell \wedge \varphi \equiv \bot$, which implies that $\ell\xi \wedge \varphi\xi \equiv \bot$. Thus, if $M \models qi(Q_{i-1}) \wedge Tr \wedge m'_{sk}$ then $M \not\models \varphi\xi$. Therefore, $\textsc{pMbp}(X' \cup SK, Tr \wedge m'_{sk}, M) \neq (\varphi\xi, \_)$. Meaning, once the POB that generated the lemma was blocked, it cannot be rediscovered as a predecessor of $\langle m, \xi, i\rangle$. Since the first two arguments of pMBP are the same in all applications of **Predecessor** on $\langle m, \xi, i\rangle$ (due to the deterministic skolemization), the finite range of pMBP implies that only finitely many predecessors are generated for the POB $\langle m, \xi, i\rangle$. □

Thus, for any value of N, there is only a finite number of POBs that are added to $\mathcal{Q}$ and processed by the rules, resulting in a finite number of rule applications. Moreover, since `Quic3_Push` restricts the use of quantified lemmas to existing ground instances and a finite instantiation scheme, and since the other rules also use only these instances, all satisfiability queries posed to the solver are of quantifier-free formulas in the combined theories of LIA and Arrays, and as a result guaranteed to terminate. This means that each rule is terminating. Therefore, `Quic3_Main` always makes progress in the following sense:

Lemma 6. *For every $k \in \mathbb{N}$, `Quic3_Main` either reaches $N = k$, returns* Safe*,, or finds a counterexample.*

Shortest Counterexamples. `Quic3_Main` increases N only after an interpolation sequence of length N is obtained, in which case it is guaranteed that no counterexample up to this length exists. Combined with Lemma 6 that ensures progress, this implies that `Quic3_Main` always find a shortest counterexample, if one exists:

Corollary 1. *If there exists a counterexample, then `Quic3_Main` is guaranteed to terminate and return a shortest counterexample.*

5 Quantified Generalization

QUIC3 uses quantified POBs to generate quantified lemmas. However, these lemmas are sometimes too specific, hindering convergence. This is addressed by *quantified generalization* (**QGen**), a key part of QUIC3. The QUIC3 rules in Alg. 1 are extended with the rule **QGen** shown in Alg. 2, and `Quic3_MakeSafe` (Figure 3) is extended with a call to `QGen`, which implements **QGen**, before a new lemma is added to its corresponding frame.

***QGen** rule.* **QGen** generalizes a (potentially quantified) lemma $(\ell, \xi) \in Q_{i+1}$ into a new quantified lemma (g, σ) such that $(\forall g) \rightarrow (\forall \ell)$ is valid, i.e., the new lemma g

QGen For $0 \leq i < n$ and a lemma $(\ell, \xi) \in Q_{i+1}$, let g be a formula and σ a substitution such that (i) $g\sigma \equiv \ell\xi$, (ii) $FVars(\ell) \subseteq FVars(g)$, and (iii) $\mathcal{F}(qi(Q_i)) \to \forall g'$. Then, add (g, σ) to Q_j for all $0 \leq j \leq i+1$.

Algorithm 2: QGen rule for Quantified Generalization in QUIC3.

is stronger than ℓ. The new quantified lemma g and a substitution ρ (s.t. $g\rho \equiv \ell$) are constructed by abstracting some terms of ℓ with fresh universally quantified variables. If the new formula $\forall g$ is a valid lemma, i.e., $\mathcal{F}(qi(Q_i)) \to \forall g'$ is valid, then **QGen** adds (g, σ) to Q_j for $0 \leq j \leq i+1$, where $\sigma = \xi|\rho$. Note that the check ensures that the new lemma maintains the interpolation sequence property of the trace. In the rest of this section, we describe two heuristics to implement **QGen** that we found useful in our benchmarks.

*Simple **QGen*** abstracts a single term in the input lemma ℓ by introducing one *additional* universally quantified variable to ℓ. In the new lemma g, the new variable v appears only as an index of an array (e.g., $\mathsf{sel}(A, v)$) or as an offset (e.g., $\mathsf{sel}(A, i+v)$). Simple **QGen** considers all sel terms in ℓ and identifies sub-terms t of index terms for which ℓ imposes lower and upper bounds. Each term t is abstracted in turn with bounds used as guards. For example, if ℓ is $0 < sz \to (\mathsf{sel}(A, 0) = 42)$ and $t = 0$ of $\mathsf{sel}(A, 0)$, then a candidate (g, σ) is $0 \leq v_0 < sz \to \mathsf{sel}(A, v_0) = 42$, and $\{v_0 \mapsto 0\}$, where v_0 is universally quantified.

*Arithmetic **QGen***. Simple **QGen** does not infer correlations neither between abstracted terms nor between index and value terms. For example, it is unable to create a lemma of the form $\forall v \cdot 0 \leq v < sz \to (\mathsf{sel}(A, v) = exp(v))$, where $exp(v)$ is some linear expression involving v. *Arithmetic* **QGen** addresses this limitation by extracting and generalizing a correlation between interpreted constants in the input lemma ℓ. Arithmetic **QGen** works on lemmas ℓ of the form $(\psi \wedge \phi_0 \wedge \cdots \wedge \phi_{n-1}) \to \phi_n$, where there is a formula $p(\boldsymbol{v})$ with free variables $\boldsymbol{v}$ and a set of substitutions $\{\sigma_k\}_{k=0}^{n}$ s. t. $\phi_k = p\sigma_k$. For example, ℓ is $((1 < sz) \wedge (\mathsf{sel}(A, 0) = 42)) \to (\mathsf{sel}(A, 1) = 44)$, where $p(i, j)$ is $\mathsf{sel}(A, i) = j$, σ_0 is $\{i \mapsto 0, j \mapsto 42\}$, and σ_1 is $\{i \mapsto 1, j \mapsto 44\}$. The substitutions can be viewed as *data points* and generalized by a convex hull, denoted ch. For example, $ch(\{\sigma_0, \sigma_1\})$ = $0 \leq i \leq 1 \wedge j = 2i + 42$. The lemma ℓ is strengthened by replacing the substitution of ϕ_n with the convex hull by rewriting ℓ into $\forall \boldsymbol{v} \cdot (ch(\{\sigma_1, \ldots, \sigma_n\}) \wedge \psi \wedge \phi_0 \cdots \wedge \phi_{n-1}) \to p(\boldsymbol{v})$. In our running example, this generates $\forall i, j \cdot (0 \leq i \leq 1 \wedge j = 2i + 42 \wedge 1 < sz) \wedge (\mathsf{sel}(A, 0) = 42)) \to (\mathsf{sel}(A, i) = j)$. Note that only ϕ_n is generalized, while all other ϕ_k, $0 \leq k < n$, provide the data points. Applying standard generalization might simplify the lemma further by dropping $(\mathsf{sel}(A, 0) = 42)$ and combining $i \leq 1 \wedge 1 < sz$ into $1 < sz$, resulting in $\forall i \cdot (0 \leq i \leq sz) \to (\mathsf{sel}(A, i) = 2i + 42)$. Note that arithmetic **QGen** applies to arbitrary linear arithmetic terms by replacing the convex hull (ch) with the polyhedral join ($\sqcup$).

These two generalizations are sufficient for our benchmarks. However, the power of QUIC3 comes from the ability to integrate additional generalizations, as required. For example, arithmetic **QGen** can be extended to consider not only a single lemma, but also mine other existing lemmas for potential data points.

6 Experimental Results

We have implemented QUIC3 within the CHC engine of Z3 [12,22] and evaluated it on array manipulating C programs from SV-COMP [4] and from [13]. We have converted C programs to CHC using SEAHORN [19]. In most of these examples, array bounds are fixed constants. We have manually generalized array bounds to be symbolic to ensure that the problems require quantified invariants. Note, however, that our approach is independent of the value of the array bound (concrete or symbolic). We stress that using SEAHORN prevents us from using the "best CHC encoding" for a given problem, which is unfortunately a common evaluation practice. By using SEAHORN as is, we show how QUIC3 deals with complex realistic intermediate representation. For example, SEAHORN generates constraints supporting memory allocation and pointer arithmetic. This complicates the necessary inductive invariants even for simple examples. While we could have used a problem-specific encoding for specially selected benchmarks, such an encoding does not uniformly extend to all SV-COMP benchmarks.

Experiments were done on a Linux machine with an Intel E3-1240V2 CPU and a timeout of 300 seconds. The source code for QUIC3 is available in the main Z3 repository at `https://github.com/Z3Prover/z3`. The CHC for all the benchmarks are available at `https://github.com/chc-comp/quic3`. The results for the safe instances – the most interesting – are shown in Table 1. We compare with the SPACER engine of Z3. SPACER supports arrays, but not quantifiers. As expected, SPACER times out on all of the benchmarks. We emphasize the difference in the number of lemmas discovered by both procedures. Clearly, since QUIC3 discovers quantified lemmas, it generates significantly fewer lemmas than SPACER. Each quantified lemma discovered by QUIC3 represents many ground lemmas that are discovered by SPACER.

As shown in Table 1, QUIC3 times out on some of the instances. This is due to a deficiency of the current implementation of **QGen**. Currently, **QGen** only considers one candidate for abstraction, and generalization fails if that candidate fails. Allowing **QGen** to try several candidates should solve this issue.

Unfortunately, we were unable to compare QUIC3 to other related approaches. To our knowledge, tools that participated in SV-COMP 2018 are not able to discover the necessary quantified invariants and often use unsound (i.e., bounded) inference. The closely related tools, including SAFARI [1], BOOSTER [2], and [13] are no longer available. Based on our understanding of their heuristics, the invariants required in our benchmarks are outside of the templates supported by these heuristics.

7 Related Work

Universally quantified invariants are necessary for verification of systems with unbounded state size (i.e., the size of an individual system state is unbounded) such as array manipulating programs, programs with dynamic memory allocation, and parameterized systems in general. Thus, the problem of universal invariant inference has been a subject of intense research in a variety of areas of automated verification. In this section, we present the related work that is technically closest to ours and is applicable to the area of software verification.

Table 1. Summary of results. TO is timeout; *Depth* is the size of inductive trace; *Lemmas* and *Inv* are the number of lemmas discovered overall and in invariant, respectively.

Benchmark	QUIC3				Z3/SPACER	
	Depth	Lemmas	Inv	Time [s]	Depth	Lemmas
array-init-const	6	24	7	0.14	130	4,483
array-init-partial	9	45	12	0.34	126	4,224
array-mono-set	6	25	9	0.22	70	2,436
array-mono-tuc	6	25	9	0.21	70	2,422
array-mul-init-tuc	129	8,136	–	TO	131	8,393
array-nd-2-c-true	6	37	–	TO	39	1,482
array-reverse	6	21	5	0.18	144	729
array-shadowinit-tuc	30	252	–	TO	99	5,005
array-swap	13	136	64	6.38	45	2,700
array-swap-twice	14	155	–	TO	45	2,991
sanfoundry-02-tucg	11	89	31	1.57	46	1,986
sanfoundry-10-tucg	11	71	23	0.67	109	3,245
sanfoundry-27-tucg	6	24	7	0.14	131	4,568
std-compMod-tucg	10	120	61	5.48	58	3,871
std-copy1-tucg	6	33	14	0.33	89	4,035
std-copy2-tucg	9	65	25	0.77	73	2,751
std-copy3-tucg	13	109	39	1.86	76	2,806
std-copy4-tucg	18	217	–	TO	85	3,416
std-copy5-tucg	19	233	76	5.47	90	3,642
std-copy6-tucg	22	301	–	TO	97	3,991
std-copy7-tucg	25	357	–	TO	101	4,321
std-copy8-tucg	27	430	105	8.05	106	4,581

Benchmark	QUIC3				Z3/SPACER	
	Depth	Lemmas	Inv	Time [s]	Depth	Lemmas
std-copy9-tucg	31	538	145	14.74	111	5,078
std-copyInitSum2-tucg	32	511	–	TO	77	2,987
std-copyInitSum3-tucg	14	127	–	TO	76	3,103
std-copyInitSum-tucg	9	59	21	0.43	78	3,085
std-copyInit-tucg	10	69	27	0.59	75	2,851
std-find-tucg	8	35	7	0.32	105	2,915
std-init2-tucg	7	29	8	0.14	88	3,662
std-init3-tucg	7	30	8	0.14	95	4,122
std-init4-tucg	7	31	8	0.14	94	3,898
std-init5-tucg	7	32	8	0.14	93	4,152
std-init6-tucg	7	33	8	0.15	95	4,090
std-init7-tucg	7	34	8	0.14	100	4,916
std-init8-tucg	7	35	8	0.15	97	4,604
std-init9-tucg	7	32	11	0.21	100	4,929
std-maxInArray-tucg	7	30	9	0.33	132	4,618
std-minInArray-tucg	7	30	10	0.27	133	4,686
std-palindrome-tucg	5	14	–	TO	64	1,717
std-part-orig-tucg	10	83	11	11.59	138	5,035
std-part-tucg	13	103	41	1.7	132	4,746
std-sort-N-nd-assert-L	12	100	15	5.02	5	17
std-vararg-tucg-tt	9	40	10	0.23	133	4,622
std-vector-diff-tucg	12	112	14	2.94	76	2,964

Classical predicate abstraction [17,3] has been adapted to quantified invariants by extending predicates with *skolem* (fresh) variables [14,26]. This is sufficient for discovering complex loop invariants of array manipulating programs similar to the ones used in our experiments. These techniques require a decision procedure for satisfiability of universally quantified formulas, and, significantly complicate predicate discovery (e.g., [27]). QUIC3 extends this work to the IC3 framework in which the predicate discovery is automated and quantifier instantiation and instance discovery are carefully managed throughout the procedure.

Recent work [7,29,20] studies this problem via the perspective of discovering universally quantified models for CHCs. These works show that fixing the number of expected quantifiers in an invariant is sufficient to approximate quantified invariants by discovering a quantifier free invariant of a more complex system. The complexity comes in a form of transforming linear CHC to non-linear CHC (*linear* refers to the shape of CHC, not the theory of constraints). Unlike predicate abstraction, guessing the predicates apriori is not required. However, both the quantifiers and their instantiations are guessed eagerly based on the syntax of the input problem. In contrast, QUIC3 works directly on linear CHC (i.e., a transition system), and discovers quantifiers and instantiations on demand. Hence, QUIC3 is not limited to a fixed number of quantifiers, and, unlike these techniques, is guaranteed to find the shortest counterexample.

Model-Checking Modulo Theories (MCMT) [16] extends model checking to array manipulating programs and has been used for verifying heap manipulating programs and parameterized systems (e.g., [11]). It uses a combination of quantifier elimination (QELIM) for computing predecessors of Bad, satisfiability checking of universally quantified formulas for pruning exploration (and convergence check), and custom generalization heuristics. In comparison, QUIC3 uses MBP instead of QELIM and uses generalizations based on bounded exploration.

SAFARI [1] (and later BOOSTER [2]), that extend MCMT with Lazy Abstraction With Interpolation (LAWI) [28], is closest to QUIC3. As in LAWI, interpolation (in case of SAFARI, for the theory of arrays [10]) is used to construct a quantifier-free proof π of bounded safety. The proof π is generalized by universally quantifying out some terms, and a decision procedure for universally quantified formulas is used to determine convergence. The key differences between SAFARI and QUIC3 are the same as between LAWI and IC3. We refer the reader to [30] for an in-depth comparison. Specifically, `Quic3_MakeSafe` computes an interpolation sequence that can be used for SAFARI. However, unlike SAFARI, QUIC3 does not rely on an external array interpolation procedure. Moreover, in QUIC3, the generalizations are dynamic and the quantifiers are introduced as early as possible, potentially exponentially simplifying the bounded proof. Finally, QUIC3 manages its quantifier instantiations to avoid relying on an external (semi) decision procedure. The acceleration techniques used in BOOSTER are orthogonal to QUIC3 and can be combined in a form of pre-processing.

To our knowledge, UPDR [23] is the only other extension of IC3 to quantified invariants. The key difference is that UPDR focuses on programs specified using the Effectively PRopositional (EPR) fragment of *uninterpreted* first order logic (e.g., without arithmetic) for which quantified satisfiability is decidable. As such, UPDR does not deal with quantifier instantiation and its mechanism for discovering quantifiers is different. UPDR is also limited to abstract counterexamples (i.e., counterexamples to existence of universal inductive invariants, as opposed to counterexamples to safety).

Interestingly, QUIC3 is closely related to algorithms for quantified satisfiability (e.g., [9,15,6]). QUIC3 uses a MBP to construct a complete instantiation, if possible. However, unlike [9,15], the convergence (of `Quic3_MakeSafe`) does not rely on any syntactic feature of the quantified formula.

8 Conclusion

In this paper, we present QUIC3, an extension of IC3 to reasoning about array manipulating programs by discovering quantified inductive invariants. While our extension keeps the basic structure of the IC3 framework, it significantly affects how lemmas and proof obligations are managed and generalized. In particular, guaranteeing progress in the presence of quantifiers requires careful management of the necessary instantiations. Furthermore, discovering quantified lemmas, requires new lemma generalization techniques that are able to infer universally quantified facts based on several examples. Unlike previous works, our generalizations and instantiations are done *on demand* guided by the property and current proof obligations. We have implemented QUIC3 in the CHC engine of Z3 and show that it is competitive for reasoning about C programs.

Acknowledgements. This publication is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). The research was partially supported by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University, and the United States-Israel Binational Science Foundation (BSF) grants No. 2016260 and 2012259. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPAS-2017-507912.

References

1. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *CAV*, 2012.
2. F. Alberti, S. Ghilardi, and N. Sharygina. Booster: An acceleration-based verification framework for array programs. In *ATVA*, 2014.
3. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *TACAS*, 2001.
4. D. Beyer. Software verification with validation of results - (report on SV-COMP 2017). In *TACAS*, 2017.
5. N. Bjørner and A. Gurfinkel. Property directed polyhedral abstraction. In *VMCAI'15*, 2015.
6. N. Bjørner and M. Janota. Playing with quantified satisfaction. In *LPAR*, 2015.
7. N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *Static Analysis (SAS)*, 2013.
8. A. R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, 2011.
9. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2006.
10. R. Bruttomesso, S. Ghilardi, and S. Ranise. Quantifier-free interpolation of a theory of arrays. *Logical Methods in Computer Science*, 8(2), 2012.
11. S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Invariants for finite instances and beyond. In *FMCAD*, 2013.
12. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.
13. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *European Symposium on Programming (ESOP)*, 2010.
14. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, 2002.
15. Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification (CAV)*, 2009.
16. S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In *IJCAR'10*, 2010.
17. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV'97*, 1997.
18. A. Gurfinkel and A. Ivrii. Pushing to the top. In *FMCAD*, 2015.
19. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *Computer Aided Verification (CAV)*, 2015.
20. A. Gurfinkel, S. Shoham, and Y. Meshman. SMT-based verification of parameterized systems. In *FSE*, 2016.
21. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT'12*, 2012.
22. K. Hoder, N. Bjørner, and L. M. de Moura. μZ- an efficient engine for fixed points with constraints. In *Computer Aided Verification (CAV)*, 2011.
23. A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham. Property-directed inference of universal invariants or proving their absence. In *CAV*, 2015.
24. A. Komuravelli, N. Bjørner, A. Gurfinkel, and K. L. McMillan. Compositional verification of procedural programs using Horn clauses over integers and arrays. In *FMCAD*, 2015.
25. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *Computer Aided Verification (CAV)*, 2014.
26. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, 2004.
27. S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *Computer Aided Verification (CAV)*, 2004.
28. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
29. D. Monniaux and L. Gonnord. Cell morphing: From array programs to array-free Horn clauses. In *Static Analysis (SAS)*, 2016.
30. Y. Vizel and A. Gurfinkel. Interpolating property directed reachability. In *CAV'14*, 2014.