Second Order Quantifier Elimination: Towards Verification Applications

Silvio Ghilardi[0000−0001−6449−6883] (✉) and Elena Pagani[0000−0001−7162−5997]

Università degli Studi di Milano, Milano, Italy
{silvio.ghilardi, elena.pagani}@unimi.it

Abstract. We develop *quantifier elimination procedures* for a fragment of higher order logic arising from the formalization of distributed systems (especially of fault-tolerant ones). Such procedures can be used in symbolic manipulations like the computation of Pre/Post images and of projections. We show in particular that our procedures are quite effective in producing *counter abstractions* that can be model-checked using standard SMT technology.

1 Introduction

Building accurate *declarative* models of distributed systems requires some complex logic, because integer and boolean variables are not sufficient: since such systems are *parameterized* (i.e. they are composed by a finite but unspecified number of processes), one needs to use arrays [1, 17] and, in the fault-tolerant case, also cardinality constraints for arrays [3, 4, 6, 12]. Since arrays are modeled by function symbols, when symbolic manipulations require to eliminate them, some form of higher-order quantifier elimination or of higher order abstraction is needed. Although in many situations existentially quantified array variables can be eliminated via explicit definitions (see the implementations in [9, 18]), this is no longer the case for concurrent and reactive systems exhibiting a large degree of non determinism.

Quantifier elimination is a rare phenomenon in second order logic, but not completely unexpected, witness the large literature on correspondence theory in modal logic. For our intended applications, some quantifier elimination results were already mentioned in [4] (Section 7.1, Thm 4) and a preliminary implementation is already available [15]. In this paper, we extend the results from our previous paper [4] by obtaining quantifier elimination for formulae containing matrices (i.e. binary arrays) and, more important, by covering formulae having *an extra universal quantifier* (Theorems 2 and 3 below). Such expansions allow us to produce arithmetic projections of more systems and to analyze benchmarks already covered in [15] in a more fine-grained way, so as to better match the pseudo-code specifications of the original papers from the distributed algorithms literature.

In: P. Koopmann, S. Rudolph, R. Schmidt, C. Wernhard (eds.): *SOQE 2017 – Proceedings of the Workshop on Second-Order Quantifier Elimination and Related Topics, Dresden, Germany, December 6–8, 2017*, published at http://ceur-ws.org.

The paper is organized as follows: in Section 2 we introduce preliminary notation for higher order logic, in Section 3 we describe our quantifier elimination results and in Section 4 we show how to apply the results to verification problems. This paper is focused on algorithmic procedures; the reader can find the detailed analysis of a benchmark in an Appendix of the extended (online available) version [16] (more examples, analyzed with the same methods but requiring much weaker quantifier elimination results,[1] can be found in [15], where also some experiments are reported).

2 Higher Order Logic and Flat Constraints

In order to have enough expressive power, we use higher order logic, more specifically *Church's type theory* (see e.g. [5] for an introduction to the subject).[2] It should be noticed, however, that our primary aim is *to supply a framework for model-checking and not to build a deductive system.* Thus we shall introduce below only suitable languages (via higher order signatures) and a semantics for such languages - such semantics can be specified e.g. inside any classical foundational system for set theory. In addition, as typical for model-checking, we want to constrain our semantics so that certain sorts have a fixed meaning: the primitive sort $\mathbb{Z}$ has to be interpreted as the (standard) set of integers, the sort Ω has to be interpreted as the set of truth values $\{\mathtt{tt}, \mathtt{ff}\}$; moreover, some primitive sorted operations like $+, 0, S$ (addition, zero, successor for natural numbers) and $\wedge, \vee, \rightarrow, \neg$ (Boolean operations for truth values) must have their natural interpretation. Some sorts might be *enumerated*, i.e. they must be interpreted as a specific finite 'set of values' $\{\mathtt{a}_0, \ldots, \mathtt{a}_k\}$, where the $\mathtt{a}_i$'s are mentioned among the constants of the language and are assumed to be distinct. Finally, we may ask for a primitive sort to be interpreted as a *finite set* (by abuse, we shall call such sorts *finite*): for instance, we shall constrain in this way the sort `Proc` modeling the set of processes in a distributed system. In addition, if a sort is interpreted into a finite set, we may constrain some numerical parameter (usually, the parameter we choose for this is named `N`) to indicate the cardinality of such finite set. The notion of constrained signature below incorporates all the above requirements in a general framework.

A *constrained signature* Σ consists of a set of (primitive) sorts and of a set of (primitive) sorted function symbols,[3] together with a class $\mathcal{C}_\Sigma$ of Σ-structures, called the *models* of Σ.[4] Using primitive sorts, *types* can be built up using ex-

[1] Quantifier elimination required in the benchmarls analyzed in [15] is in fact essentially confined to the BAPA-fragment known since [26].

[2] Some notation we use might look slightly non-standard; it is similar to the notation of [27].

[3] These include 0-ary function symbols, called constants; constants of sort $\mathbb{Z}$ will be called (arithmetic) *parameters*.

[4] In the standard model-checking literature, $\mathcal{C}_\Sigma$ is a singleton; here we must allow *many* structures in $\mathcal{C}_\Sigma$, because our model-checking problems are *parametric*: the sort modeling the set of processes of our system specifications must be interpreted onto

ponentiation (= functions type); *terms* can be built up using variables, function symbols, as well as λ-abstraction and functional application.

Our constrained signatures always include the sort Ω of truth-values; terms of type Ω are called *formulae* (we use greek letters $\alpha, \beta, \ldots, \phi, \psi, \ldots$ for them). For a type S, the type $S \to \Omega$ is indicated as $\wp(S)$ and called the *power set* of S; if S is constrained to be interpreted as a finite set, Σ might contain a cardinality operator $\sharp : \wp(S) \longrightarrow \mathbb{Z}$, whose interpretation is assumed to be the intended one ($\sharp s$ is the number of the elements of s - as such it is always a nonnegative number). If ϕ is a formula and S a type, we use $\{x^S \mid \phi\}$ or just $\{x \mid \phi\}$ for $\lambda x^S \phi$. We assume to have binary equality predicates for each type; universal and existential quantifiers for formulæ can be introduced by standard abbreviations (see e.g. [27]). We shall use the roman letters $x, y, \ldots, i, j, \ldots, v, w, \ldots$ for variables (of course, each variable is suitably typed, but types are left implicit if confusion does not arise). Bold letters like $\mathbf{v}$ (or underlined letters like $\underline{x}$) are used for tuples of free variables; below, we indicate with $t(\mathbf{v})$ the fact that the term t has free variables included in the list $\mathbf{v}$ (whenever this happens, we say that t is a $\mathbf{v}$-*term*, or a $\mathbf{v}$-*formula* if it has type Ω). The result of a simultaneous substitution of the tuple of variables $\mathbf{v}$ by the tuple of (type matching) terms $\underline{u}$ in t is denoted by $t(\underline{u}/\mathbf{v})$ or directly as $t(\underline{u})$.

Given a tuple of variables $\mathbf{v}$, a Σ-*interpretation* of $\mathbf{v}$ in a model $\mathcal{M} \in \mathcal{C}_\Sigma$ is a function $\mathcal{I}$ mapping each variable onto an element of the correponding type (as interpreted in $\mathcal{M}$). The evaluation of a term $t(\mathbf{v})$ according to $\mathcal{I}$ is recursively defined in the standard way and is written as $t_{\mathcal{M},\mathcal{I}}$. A Σ-formula $\phi(\mathbf{v})$ is *true* under $\mathcal{M}, \mathcal{I}$ iff it evaluates to $\mathtt{tt}$ (in this case, we may also say that $\mathbf{v}_{\mathcal{M},\mathcal{I}}$ *satisfies* ϕ); ϕ is *valid* iff it is true for all models $\mathcal{M} \in \mathcal{C}_\Sigma$ and all interpretations $\mathcal{I}$ of $\mathbf{v}$ over $\mathcal{M}$. We write $\models_\Sigma \phi$ (or just $\models \phi$) to mean that ϕ is valid and $\phi \models_\Sigma \psi$ (or just $\phi \models \psi$) to mean that $\phi \to \psi$ is valid; we say that ϕ and ψ are Σ-*equivalent* (or just equivalent) iff $\phi \leftrightarrow \psi$ is valid.

2.1 Flat Cardinality Constraints

Let us fix a constrained signature Σ for the remaining part of the paper. Such Σ should be adequate for modeling parameterized systems, hence we assume that Σ consists of:

(i) the integer sort $\mathbb{Z}$, together with some parameters (i.e. free individual constants) as well as all operations and predicates of linear arithmetic (namely, $0, 1, +, -, =, <, \equiv_n$);
(ii) the enumerated truth value sort Ω, with the constants $\mathtt{tt}, \mathtt{ff}$ and the Boolean operations on them;

---

a finite set whose cardinality is not a priori fixed. Our definition of a 'constrained signature' is analogous to the definition of a 'theory' in SMT literature; in fact, in SMT literature, a 'theory' is just a pair given by a signature and a class of structures. When transferred to a higher order context, such definition coincides with that of a 'constrained signature' above (thus our formal preliminary definitions are very similar to e.g. that of [29]).

(iii) a finite sort `Proc`, whose cardinality is constrained to be equal to the arithmetic parameter `N` (in the applications, this sort is used to represent the processes acting in our distributed systems);

(iv) a further sort `Data`, with appropriate operations, modeling local data; we assume that (a) *first-order quantifier elimination* holds for `Data`, meaning that all first-order formulæ built up from `Data`-atoms (i.e. from variables of type `Data` using operations and predicates relative to the sort `Data`) are equivalent to quantifier-free ones; (b) *ground* (i.e. variable-free) `Data`-atoms are equivalent to $\bot$ or to $\top$.

In principle, we could consider having finitely many signatures for data instead of just one, but this generalization is only apparent because one can use product sorts and recover component sorts via suitable pairing and projection operations.

If `Data` is an enumerated sort, we call Σ *finitary*; the subsignature Σ_0 of Σ obtained by restricting to sorts and operations in (i)-(ii) is called the *arithmetic* subsignature of Σ.

In the syntactic definitions below, we freely take inspiration from [3], however the present framework is greatly simplified because we do not view `Proc` as a subsort of $\mathbb{Z}$, like in [3]; in addition, notice that Σ does not contain operations or relation symbols specific to the sort `Proc` (apart from equality) - this restriction reduces terms of sort `Proc` to just variables.

Below, besides *integer* variables (namely variables of sort $\mathbb{Z}$), *data* variables (namely variables of sort `Data`) and *index* variables (namely variables of sort `Proc`), we use two other kinds of variables, that we call *array-ids* and *matrix-ids*. An array-id is a variable of type $\texttt{Proc} \to \texttt{Data}$ or of type $\texttt{Proc} \to \mathbb{Z}$ and a matrix-id is a variable of type $\texttt{Proc} \to (\texttt{Proc} \to \texttt{Data})$ or of type $\texttt{Proc} \to (\texttt{Proc} \to \mathbb{Z})$. Array-ids and matrix-ids of codomain sort $\mathbb{Z}$ are called *arithmetical* array-ids or matrix-ids; if `Data` is enumerated, array-ids and matrix-ids of codomain sort `Data` are called *finitary*. If M is a matrix-id and i, y are index variables, we may write $M_i(y)$ or $M(i, y)$ instead of $M(i)(y)$.

Let us now introduce some useful classes of formulæ.

- *Open formulæ*: these are built up from atomic formulæ containing arithmetic parameters and the above mentioned variables, using Boolean connectives only (no binders, i.e. no λ-abstractors and no quantifiers).
- *1-Flat formulæ*: these are formulæ of the kind $\phi(\sharp\{x \mid \psi_1\}/z_1, \ldots, \sharp\{x \mid \psi_n\}/z_n)$, where $\phi(z_1, \ldots, z_n), \psi_1, \ldots, \psi_n$ are open and x is a variable of type `Proc`.
- Given an index variable i, a formula ϕ is said to be *i-uniform* with respect to a matrix-id M (resp. an array-id a) iff i is not used as a bounded variable in ϕ and the only terms occurring in ϕ containing an occurrence of M (resp. of a) are of the kind $M_i(y)$ (resp. $a(i)$) for a variable y.

Notice that, some quantified formulæ can be rewritten as 1-flat formulæ: for instance $\forall i\ (a(i) = c \rightarrow b(i) = d)$ is the same as $\sharp\{i \mid a(i) = c \rightarrow b(i) = d\} = \mathtt{N}$,[5] and similarly $\exists i\ (a(i) = c)$ can be re-written as $\sharp\{i \mid a(i) = c\} > 0$.

Remark 1. 1-Flat formulæ of this paper are slightly different from the flat formulæ of [3, 4] (they roughly correspond to the flat formulæ of degree 1 of [4]); the definition here is not recursive and is simplified by the fact that we do not have nonvariable terms of type `Proc`; on the other hand, we allow matrix-ids to occur in our formulæ, whereas the syntax of [3, 4] is restricted to array-ids.

3 Quantifier Elimination

In this technical section we state and prove the quantifier elimination results we need. Let us fix a constrained signature Σ like in Subsection 2.1. We first investigate in a closer way our open formulæ. Notice first that if an open formula is *pure* (i.e. it does not contain array-ids or matrix-ids), then it is a Boolean combination of arithmetic, index or data atoms, where:

- arithmetic atoms are built up from variables of sort $\mathbb{Z}$, parameters (i.e free constants of sort $\mathbb{Z}$), by using $=, <, \equiv_n$ as predicates and $+, -, 0, 1$ as function symbols;
- index atoms are of the kind $i = j$, where i, j are variables of sort `Proc` (we do not consider further operations and predicates for this sort - apart from equality - in this paper);
- data atoms are built up from variables of sort `Data` by applying some specific set of predicates and operations (predicates include equality, all arguments of such predicates and operations are of type `Data`).

By assumption (see Subsection 2.1), quantifier elimination holds for first-order `Data`-formulæ, but this result extends very easily to all pure first-order formulæ. We state this formally as a Lemma:

Lemma 1. *Any pure first-order formula is equivalent to an open pure first-order formula.*

Proof. Using prenex formula transformations, it is sufficient to show how to eliminate a quantifier $\exists x\, \alpha$, where α is open and pure. Actually, using disjunctive normal forms, we can assume that α is a conjunction of literals. Pushing the existential quantifier inside, we can assume that such literals are all arithmetic, all index or all data literals, depending on the sort of x. The case of arithmetic literals is covered by Presburger quantifier elimination [28], whereas the case of data literals is covered by our assumption. It remains to consider the case of index literals; excluding trivial cases where the existential quantifier is redundant or eliminable by substitution, we are left with the case where α is $x \neq y_1 \wedge \cdots \wedge x \neq$

[5] Strictly speaking, this formula is 1-flat only after a bound variable renaming (we need to rename i to x). We always feel free to apply such α-conversions in the paper.

y_n. By introducing a disjunction of cases (and by distributing the existential quantifier over such disjunction and removing redundant variables), we reduce to a disjunction of formulæ of the kind

$$\exists x\ (x \neq y_1 \wedge \cdots \wedge x \neq y_{n'} \wedge \bigwedge_{i \neq j} y_i \neq y_j)$$

The latter is equivalent to $\mathtt{N} > \bar{n}'$, where $\bar{n}'$ is $1 + \cdots + 1$ (n'-times). ⊣

In case array-ids and matrix-ids do not occur, 1-flat formulæ can also be trivialized:[6]

Lemma 2. *A 1-flat formula without array-ids and matrix-ids is equivalent to a pure formula.*

Proof. Let us eliminate subterms t of the kind $\sharp\{x \mid \alpha\}$ (with pure α) inside a pure formula ϕ. We can first remove from α arithmetic and data atoms, as well as index atoms not containing x, by the following equivalence (let A be the atom to be removed):

$$\phi \leftrightarrow ([A \wedge \phi(\top/A)] \vee [\neg A \wedge \phi(\bot/A)]) \quad .$$

By Venn regions decomposition, we can assume that α is a conjunction of literals. In addition, if t is of the kind $\sharp\{x \mid x = i \wedge \alpha\}$, we can remove it using the equivalence:

$$\phi \leftrightarrow ([\alpha(i/x) \wedge \phi(1/t)] \vee [\neg\alpha(i/x) \wedge \phi(0/t)]) \quad .$$

Thus we are left only with the case in which t is $\sharp\{x \mid \bigwedge_{s=1}^{n} x \neq i_s\}$; we can also assume that ϕ entails $\bigwedge_{s \neq s'} i_s \neq i_{s'}$ (otherwise we can force this by making ϕ a disjunction of case distinctions). Then we can remove t using

$$\phi \leftrightarrow (\mathtt{N} \geq \bar{n} \wedge \phi(\mathtt{N} - \bar{n}/t)) \vee (\mathtt{N} < \bar{n} \wedge \phi(0/t)) \quad .$$

Once all t are removed (one by one), the statement is proved. ⊣

It is now convenient to introduce a notation for open (not necessarily pure) formulæ (from now on we shall reserve the letters $\alpha, \beta, \ldots$ to first-order *pure* formulæ, to evidentiate them). Considering that there are no operation symbols of sort `Proc`, the only new terms that might arise in open non pure formulæ (wrt pure formulæ) are of the kind $a(i)$ or $M_i(j)$, where a is an array-id, M is a matrix-id and i, j are variables of sort `Proc`. Thus we may write an open formula ϕ as the formula obtained by replacing in a pure formula some arithmetic variables with terms of the kind $a(i)$ or $M_i(j)$. If our open ϕ does not contain matrix-ids, we can write it as

$$\alpha(\underline{z}, \underline{k}, \mathbf{a}(\underline{k})/\underline{e}, \underline{d}) \quad \text{or simply as } \alpha(\underline{z}, \underline{k}, \mathbf{a}(\underline{k}), \underline{d}) \tag{1}$$

[6] If the sort `Proc` is identified with a definable finite subset of $\mathbb{Z}$, the result still holds but is much less trivial: to get it, one must apply results from Presburger arithmetic with counting quantifiers [30].

where $\alpha(\underline{z}, \underline{k}, \underline{e}, \underline{d})$ is pure, $\underline{z}$ is a tuple of arithmetic variables, $\underline{k}$ is a tuple of index variables, $\underline{d}$ is a tuple of data variables, $\mathbf{a}$ is a tuple of array-ids (the $\underline{e}$ might be arithmetic or `Data`-variables depending on the types of the $\mathbf{a}$); if $\mathbf{a} = a_1, \ldots, a_n$ and $\underline{k} = k_1, \ldots, k_m$, then $\mathbf{a}(\underline{k})$ is the tuple

$$a_1(k_1), \ldots, a_1(k_m), \ldots, a_n(k_1), \ldots, a_n(k_m)$$

so that the matching tuple of data variables $\underline{e}$ must be indexed as $e_{11}, \ldots, e_{nm}$.

A 1-flat formula without matrix-ids is then written as

$$\alpha(\underline{z}, \underline{k}, \mathbf{a}(\underline{k}), \underline{d}, \sharp\{x \mid \beta_1(\underline{z}, x, \underline{k}, \mathbf{a}(x), \mathbf{a}(\underline{k}), \underline{d})\}, \ldots, \sharp\{x \mid \beta_s(\underline{z}, x, \underline{k}, \mathbf{a}(x), \mathbf{a}(\underline{k}), \underline{d})\})$$

or (with some abuse of notation) shortly as

$$\alpha(\underline{z}, \underline{k}, \mathbf{a}(\underline{k}), \underline{d}, \sharp\{x \mid \underline{\beta}(\underline{z}, x, \underline{k}, \mathbf{a}(x), \mathbf{a}(\underline{k}), \underline{d})\}) \tag{2}$$

where $\underline{\beta}$ is a tuple of formulæ (we use the convention that $\sharp\{x \mid \underline{\beta}\}$ stands for the tuple of terms $\sharp\{x \mid \beta_1\}, \ldots, \sharp\{x \mid \beta_s\}$). Displaying 1-flat formulæ with matrix-ids requires an even more complex notation, that we won't use though. These notations are apparently cumbersome but have the merit of displaying the essential information on how our formulæ are built up from pure formulæ.

We now state a first quantifier elimination result (this is Theorem 4 from [4], we nevertheless report the proof in an appendix of the extended version [16] of the present paper):

Theorem 1. *Suppose that ϕ is a 1-flat formula containing the array ids $\mathbf{a}, \mathbf{a}'$ (and not containing matrix-ids); then the formula $\exists \mathbf{a}'\, \phi$ is equivalent to a formula $\exists \underline{e}\, \psi$, where the $\underline{e}$ are arithmetic and data variables, ψ is 1-flat and contains only the array-ids $\mathbf{a}$.*

The following Corollary follows from Theorem 1 and Lemmas 2,1:

Corollary 1. *Suppose that ϕ is a 1-flat formula containing the array ids $\mathbf{a}$ (and not containing matrix-ids); then the formula $\exists \mathbf{a}\, \phi$ is equivalent to an open pure formula.*

Notice that the above result (as it happens with all our quantifier elimination results) immediately implies that 1-flat formulæ not containing matrix-ids are decidable for satisfiability. If the sort `Data` is enumerated and all array-ids are finitary, we can improve Corollary 1 above by including an extra quantified variable (this is useful for benchmarks, see the Appendix of [16] for an example):

Theorem 2. *Let the sort* `Data` *be enumerated and let the 1-flat formula ϕ contain only the finitary array ids $\mathbf{a}$ (and no matrix-ids); then the formula*

$$\exists \mathbf{a}\, \forall i\, \exists \underline{y}\, \phi \tag{3}$$

(where i is an index variable and the $\underline{y}$ are arithmetic and data variables) is equivalent to an open pure formula.

Proof. Let $\mathtt{Data}$ be enumerated as $\{\mathtt{a}_0, \ldots, \mathtt{a}_k\}$; let $\underline{z}$ be the arithmetic variables occurring freely in (3) and let $\underline{k} = k_1, \ldots, k_n$ be the index variables occurring freely in (3) (thus i is not among the $\underline{k}$ and the $\underline{y}$ are not among the $\underline{z}$). We can assume that the $\underline{y}$ are arithmetic variables because, since $\mathtt{Data}$ is enumerated, existential data variables can be elimitated via disjunctions. For simplicity, we assume that (3) contains only one array-id, let it be a.[7]

Before working on the formula (3), it is better to make some preprocessing steps. Our final outcome will be that of producing a disjunction of existentially quantified pure formulæ logically equivalent to (3): in fact, we introduce extra existentially quantified variables to be eliminated in the very end using Lemma 1. We need also to introduce extra information to complete (3): this extra information is achieved by rewriting (3) as a disjunction (each disjunct formalizes a suitable guess) and by operating on each disjunct separately.

Concretely, we shall freely assume that $\forall i\, \exists \underline{y}\, \phi$ in (3) is of the kind

$$\begin{array}{l} \mathtt{Diff}(\underline{k}) \ \wedge \ \bigwedge_i (a(k_i) = \mathtt{a}_{l_i}) \ \wedge \bigwedge_j (u_j = \sharp\{x \mid a(x) = \mathtt{a}_j\}) \ \wedge \\ \wedge \ \forall i\, \exists \underline{y}\, \phi'(\underline{z}, \underline{y}, i, \underline{k}, a(i), \sharp\{x \mid \underline{\beta}(\underline{z}, \underline{y}, x, i, \underline{k}, a(x), a(i))\}) \end{array} \tag{4}$$

where

- the formula $\mathtt{Diff}(\underline{k})$ says that the $\underline{k}$ are pairwise distinct (i.e. it is $\bigwedge_{i \neq j} k_i \neq k_j$): this can be assumed without loss of generality, because one can guess a partition (introducing a disjunction over all partitions) and make the appropriate replacements so as to keep only one representative for each equivalence class of variables;
- since $\mathtt{Data}$ is enumerated we can guess (via a disjunction) for each k_i the $\mathtt{a}_{l_i}$ which is the value of $a(k_i)$ (then, all occurrences of the term in the remaining part of the formula can be replaced by this $\mathtt{a}_{l_i}$);
- the u_j are fresh arithmetic variables indicating the cardinality of the set of indices whose a-value is $\mathtt{a}_j$ (these u_j are the extra existentially quantified variables to be eliminated in the very end by Lemma 1);
- $\underline{\beta}$ are open formulæ as displayed and ϕ' is a 1-flat formula as displayed (notice that the terms $a(k_i)$ do not occur anymore here, because we can assume that they have been replaced by the corresponding $\mathtt{a}_{l_i}$).

We now operate further transformations on the subformula $\forall i\, \exists \underline{y}\, \phi'$: we want to show that this formula is equivalent to a 1-flat formula (hence without the quantifier $\forall i$), so that the claim of the Theorem follows from an application of Corollary 1 and Lemma 1 - by these results in fact all quantified variables in (3) can be eliminated in favor of a pure open formula in which only the $\underline{k}, \underline{z}$ occur. When manipulating $\forall i\, \exists \underline{y}\, \phi'$ below, we assume all the information we have from (4), namely that the $\underline{k}$ are all distinct and that the values of the $a(k_i)$ are known.

[7] This is without loss of generality: since $\mathtt{Data}$ is enumerated and the $\mathbf{a}$ are finitary, one may take a product of $\mathtt{Data}$ and replace the tuple $\mathbf{a}$ with a single array with values in such a product.

As a first step, we can distinguish the case in which i is equal to some of the $\underline{k}$ from the case in which it is different from all of them; in the latter case, we can also guess the value of $a(i)$. This observation shows that $\forall i\, \phi'$ is equal to the conjunction of an open formula (expressing what happens if i is equal to any of the $\underline{k}$) with the conjunctions (varying $\mathtt{a_j}$ in our enumerated data)

$$\forall i.\ \ \mathtt{Diff}(i,\underline{k}) \wedge a(i) = \mathtt{a_j} \rightarrow \exists \underline{y}\, \phi''(\underline{z},\underline{y},i,\underline{k},\sharp\{x \mid \underline{\beta}'(\underline{z},\underline{y},x,i,\underline{k},a(x))\}) \qquad (5)$$

where the $\phi'', \underline{\beta}'$ are obtained from the $\phi', \underline{\beta}$ by replacing $a(i)$ with $\mathtt{a_j}$. Again, it will be sufficient to show that (5) is equivalent to an open formula.

First observe that ϕ'' is obtained from a pure formula by replacing arithmetic variables with the terms $\sharp\{x \mid \underline{\beta}'(\underline{z},\underline{y},x,i,\underline{k},a(x))\}$; since equality is the only predicate of sort `Proc` (and there are no function symbols of sort `Proc`), the only atoms of sort `Proc` that might occur in a pure formula are of the kind $i = k_s, k_s = k_{s'}$ for some $s \neq s'$, but these can all be replaced by $\bot$ because we have $\mathtt{Diff}(i,\underline{k})$ in the antecedent of the implication of (5). As a consequence ϕ'' *can be displayed as* $\phi''(\underline{z},\underline{y},\sharp\{x \mid \underline{\beta}'(\underline{z},x,i,\underline{k},a(x))\})$.

A similar observation applies also to the $\underline{\beta}'$, however here we must take into consideration also atoms of the kind $x = i, x = k_s$. Thus, the $\underline{\beta}'$ are built up using Boolean conectives from atoms of the kind $x = i, x = k_s$, from arithmetic atoms $A(\underline{z},\underline{y})$ and from `Data`-atoms that might contain the term $a(x)$. We can disregard arithmetic atoms, because for each such atom $A(\underline{z},\underline{y})$ we may rewrite ϕ'' as

$$[A(\underline{z},\underline{y}) \wedge \phi''(\underline{z},\underline{y},\sharp\{x \mid \underline{\beta}'(\top/A)\})] \ \vee\ [\neg A(\underline{z},\underline{y}) \wedge \phi''(\underline{z},\underline{y},\sharp\{x \mid \underline{\beta}'(\bot/A)\})]\ . \qquad (6)$$

Thus the $\underline{\beta}'$ can be displayed as $\underline{\beta}'(x,i,\underline{k},a(x))$.

When $x = i$ or $x = k_s$ (for some s) the $\underline{\beta}'$ can be simplified to $\top$ or $\bot$ because we know the values of $a(i), a(k_s)$ (and as a consequence the numbers $\sharp\{x \mid x = i \wedge \underline{\beta}'\}, \sharp\{x \mid x = k_s \wedge \underline{\beta}'\}$ are 0/1-tuples). In conclusion we have that, for some tuple of numbers $\underline{m}$[8] that can be computed, we have that (5) is equivalent to

$$\forall i.\ \ \mathtt{Diff}(i,\underline{k}) \wedge a(i) = \mathtt{a_j} \rightarrow \exists \underline{y}\, \phi''(\underline{z},\underline{y},\underline{\bar{m}} + \sharp\{x \mid \mathtt{Diff}(x,i,\underline{k}) \wedge \underline{\beta}''(a(x))\}) \quad (7)$$

where $\underline{\beta}''$ is obtained from $\underline{\beta}'$ by replacing the atoms $x = i, x = k_s$ with $\bot$. Fix now some β''_s from the tuple $\underline{\beta}''$; for every enumerated data $\mathtt{a}_k$, each of the formulæ $\beta''_s(\mathtt{a}_k)$ simplify to either $\top$ or $\bot$ and, since we know that $u_k = \sharp\{x \mid a(x) = \mathtt{a}_k\}$ from (4), we can deduce that $\sharp\{x \mid \mathtt{Diff}(x,i,\underline{k}) \wedge a(x) = \mathtt{a}_k \wedge \beta''_s(\mathtt{a}_k)\}$ is equal to either 0 (in case $\beta''_s(\mathtt{a}_k)$ simplifies to $\bot$) or to $u_k - n_k$, where n_k is the number of the $\underline{k}, i$ for which we know that $a(\underline{k}), a(i)$ is equal to $\mathtt{a}_k$. As a consequence $\sharp\{x \mid \mathtt{Diff}(x,i,\underline{k}) \wedge \underline{\beta}''(a(x))\}$ is equal to $\sum_k (u_k - n_k)$ (where the sum extends to all k such that $\beta''_s(\mathtt{a}_k)$ simplifies to $\top$).

[8] This tuple depends on j, i.e. on the $\mathtt{a}_j$ used in the antecedent of (5) (we do not indicate this dependency for simplicity).

All this can be summarized by saying that we can rewrite (7) as

$$\forall i.\ \ \mathtt{Diff}(i,\underline{k}) \wedge a(i) = \mathtt{a_j} \rightarrow \exists \underline{y}\, \theta_j(\underline{y},\underline{z},\underline{u}) \tag{8}$$

where the formulæ θ_j are pure (the tuple $\underline{u}$ is the tuple of the u_j from (4)). By Presburger quantifier elimination, we can drop the $\exists \underline{y}$, thus getting

$$\forall i.\ \ \mathtt{Diff}(i,\underline{k}) \wedge a(i) = \mathtt{a_j} \rightarrow \theta'_j(\underline{z},\underline{u}) \tag{9}$$

Since now θ'_j does not contain occurrences of i, we can rewrite this as

$$\exists i\ (\mathtt{Diff}(i,\underline{k}) \wedge a(i) = \mathtt{a_j}) \rightarrow \theta'_j(\underline{z},\underline{u}) \tag{10}$$

and finally as

$$\sharp\{x \mid \mathtt{Diff}(x,\underline{k}) \wedge a(x) = \mathtt{a_j}\} > 0 \rightarrow \theta'_j(\underline{z},\underline{u}) \tag{11}$$

This is a 1-flat formula. To sum up, our original formula (3) is equivalent to a formula of the kind $\exists a\, \exists \underline{u}\, \vartheta$, where ϑ is 1-flat. Then (after swapping the quantifiers $\exists a\, \exists \underline{u}$) we can first use Theorem 1 to remove $\exists a$ and then Lemma 1 to produce an equivalent pure open formula (involving just the arithmetic variables $\underline{z}$ and the index variables $\underline{k}$). ⊣

In case we have uniformity, we can further extend the above result to cover formulæ in which arithmetic array-ids and matrix-ids occur (see again the Appendix of [16] for an example of the use of this result):

Theorem 3. *Let the sort* `Data` *be enumerated and let i be an index variable; suppose that all matrix-ids* $\mathbf{M}$ *occurring in the 1-flat formula ϕ are i-uniform and that all array-ids* $\mathbf{a}$ *occurring in ϕ are either finitary or i-uniform. Then the formula*

$$\exists \mathbf{a}\, \exists \mathbf{M}\, \forall i\, \exists \underline{y}\, \phi \tag{12}$$

(where the $\underline{y}$ are arithmetic and data variables) is equivalent to an open pure formula.

Proof. The first step is to remove $\exists M$ for each $M \in \mathbf{M}$, using uniformity. In fact, by uniformity, M occurs in ϕ only inside terms of the kind $M_i(y)$ (for some index variable y); thus, using *choice axiom* (in the form of an anti-skolemization), we can rewrite (12) as

$$\exists \mathbf{a}\, \forall i\, \exists \mathbf{b}\ \exists \underline{y}\, \phi(\cdots \mathbf{b}/\mathbf{M}_i \cdots) \tag{13}$$

and then we can swap the existential quantifiers $\exists \mathbf{b} \exists \underline{y}$ and apply Theorem 1, thus obtaining a formula of the kind $\exists \mathbf{a}\, \forall i\, \exists \underline{y}\, \exists \underline{e}\, \psi$ where the $\underline{e}$ are further arithmetic or data variables, ψ is 1-flat and contains only the array-id $\mathbf{a}$. Let us now split the $\mathbf{a}$ as $\mathbf{a}', \mathbf{a}''$, where the $\mathbf{a}''$ are i-uniform and the $\mathbf{a}'$ are finitary. We can apply the same anti-skolemization argument to the $\mathbf{a}''$ and rewrite $\exists \mathbf{a}'\, \mathbf{a}''\, \forall i\, \exists \underline{y}\, \exists \underline{e}\, \psi$ as $\exists \mathbf{a}'\, \forall i\, \exists \underline{z}\, \exists \underline{y}\, \exists \underline{e}\, \psi(\underline{z}/\mathbf{a}''(i))$, where the $\underline{z}$ are fresh arithmetic variables replacing the terms $\mathbf{a}''(i)$ in ψ. Now Theorem 2 can be used to eliminate the $\mathbf{a}'$. ⊣

4 System Specifications and Arithmetic Projections

We now go to verification applications. We summarize the essential machinery for making quantifier elimination to apply (for more information, see [15]).

Definition 1. *A* system specification $\mathcal{S}$ *is a tuple*

$$\mathcal{S} \;=\; (\Sigma, \mathbf{v}, \Phi, \iota, \tau) \tag{14}$$

where (i) Σ *is a constrained signature, (ii)* $\mathbf{v}$ *is a tuple of variables, (iii)* Φ, ι *are* $\mathbf{v}$*-formulæ, (iv)* τ *is a* $(\mathbf{v}, \mathbf{v}')$*-formula (here the* $\mathbf{v}'$ *are renamed copies of the* $\mathbf{v}$*) such that*

$$\iota(\mathbf{v}) \models_\Sigma \Phi(\mathbf{v}), \qquad \Phi(\mathbf{v}) \wedge \tau(\mathbf{v}, \mathbf{v}') \models_\Sigma \Phi(\mathbf{v}') \quad . \tag{15}$$

In the above definition, the $\mathbf{v}$ are meant to be the variables specifying the system status, ι is meant to describe initial states and τ is meant to describe the transition relation. The $\mathbf{v}$-formula Φ, as it is evident from (15), describes an invariant of the system (known to the user). Invariants are quite useful - and often essential - in concrete verification tasks, that's why we included them in Definition 1.

A *safety problem* for a system specification $\mathcal{S}$ like above is a $\mathbf{v}$-formula $\upsilon(\mathbf{v})$; the system is *safe with respect to* υ iff there is no $n \geq 0$ such that the formula

$$\iota(\mathbf{v}_0) \wedge \tau(\mathbf{v}_0, \mathbf{v}_1) \wedge \cdots \wedge \tau(\mathbf{v}_{n-1}, \mathbf{v}_n) \wedge \upsilon(\mathbf{v}_n)$$

is satisfiable.

Directly attacking safety problems for a system like (14) might be a too difficult task, that's why it is useful to replace it with a simpler system: in our applications, we shall try to replace $\mathcal{S}$ by some $\mathcal{S}'$ whose variables are all integer variables. To this aim, we 'project' $\mathcal{S}$ onto a subsystem $\mathcal{S}'$, i.e. onto a system comprising only some of the variables of $\mathcal{S}$. In order to give a precise definition of what we have in mind, we must first consider subsignatures: here a *subsignature* Σ_0 of Σ is a signature obtained from Σ by dropping some symbols of Σ and taking as Σ_0-models the class $\mathcal{C}_{\Sigma_0}$ of the restrictions $\mathcal{M}_{|\Sigma_0}$ to the Σ_0-symbols of the structures $\mathcal{M} \in \mathcal{C}_\Sigma$. The following proposition is immediate:

Proposition 1. *Let* $\mathcal{S}_0 = (\Sigma_0, \mathbf{v}_0, \Phi_0, \iota_0, \tau_0)$ *and* $\mathcal{S} = (\Sigma, \mathbf{v}, \Phi, \iota, \tau)$ *be system specifications, with respective safety problems* $\upsilon(\mathbf{v}_0)$ *and* $\upsilon(\mathbf{v})$*. Suppose that* Σ_0 *is a subsignature of* Σ *and let* $\mathbf{v} = \mathbf{v}_0, \mathbf{v}_1$*; suppose also that the following hold:*

(i) $\models_\Sigma \Phi_0(\mathbf{v}_0) \leftrightarrow \exists \mathbf{v}_1 \Phi(\mathbf{v}_0, \mathbf{v}_1)$*;*
(ii) $\models_\Sigma \iota_0(\mathbf{v}_0) \leftrightarrow \exists \mathbf{v}_1 \iota(\mathbf{v}_0, \mathbf{v}_1)$*;*
(iii) $\models_\Sigma \tau_0(\mathbf{v}_0, \mathbf{v}_0') \leftrightarrow \exists \mathbf{v}_1 \exists \mathbf{v}_1' (\Phi(\mathbf{v}_0, \mathbf{v}_1) \wedge \tau(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_0', \mathbf{v}_1'))$*;*
(iv) $\models_\Sigma \upsilon_0(\mathbf{v}_0) \leftrightarrow \exists \mathbf{v}_1 \upsilon(\mathbf{v}_0, \mathbf{v}_1)$*.*

Then if $\mathcal{S}_0$ *is safe with respect to* υ_0*, so it is* $\mathcal{S}$ *with respect to* υ*.*[9]

[9] Notice that only the right-to-left implications of (i)-(iv) are needed for the proposition to hold; however, if we have also the left-to-right implications, the system specification $\mathcal{S}_0$ is better, in a sense that can be specified formally [15] (intuitively, the system specification $\mathcal{S}_0$ would be the best approximation of $\mathcal{S}$ that we can make using only the variables $\mathbf{v}_0$).

The system specification $\mathcal{S}_0$ satisfying the above condition (i)-(iii) with respect to $\mathcal{S}$ is called the $(\Sigma_0, \mathbf{v}_0)$-projection of $\mathcal{S}$; if Σ_0 is the arithmetical subsignature of Σ and $\mathbf{v}_0$ are all the arithmetic variables of $\mathcal{S}$, $\mathcal{S}_0$ is called the *arithmetic projection* of $\mathcal{S}$.

Theorem 4. *If Φ, ι, τ do not contain matrix-ids and are of the kind $\exists k_1 \cdots \exists k_n \phi$ for a 1-flat formula ϕ and for index variables $k_1, \ldots, k_n$, then $\mathcal{S} = (\Sigma, \mathbf{v}, \Phi, \iota, \tau)$ has an (effectively computable) arithmetic projection.*

Proof. Let $\mathbf{v}$ be $\underline{z}, \mathbf{a}$, where the $\underline{z}$ are arithmetic variables and the $\mathbf{a}$ are array variables; we also abbreviate $k_1, \ldots, k_n$ as $\underline{k}$. We need to show that a formula of the kind

$$\exists \mathbf{a} \, \exists \underline{k} \; \alpha(\underline{z}, \underline{k}, \mathbf{a}(\underline{k}), \sharp\{x \mid \underline{\beta}(\underline{z}, x, \underline{k}, \mathbf{a}(x), \mathbf{a}(\underline{k}))\}) \tag{16}$$

is equivalent to a pure arithmetic formula.[10] But this is indeed the case: just swap the existential quantifiers and apply Corollary 1 and Lemma 1. The result follows because there are no ground index atoms and all ground data atoms are equivalent to $\top$ or to $\bot$, according to our assumptions from Subsection 2.1.

Next result concerns specifications using matrix-ids in a finitary signature.

Theorem 5. *Let the sort* `Data` *be enumerated and let Φ, ι, τ be disjunctions of formulæ of the kind*

$$\exists \underline{k} \; \forall i \; \exists \underline{y} \; \phi \tag{17}$$

where ϕ is 1-flat, $\underline{k}$ are index variables, $\underline{y}$ are arithmetic and data variables and i is an index variable such that all matrix variables and all non-finitary array variables from $\mathbf{v}$ are i-uniform in ϕ; then $\mathcal{S} = (\Sigma, \mathbf{v}, \Phi, \iota, \tau)$ has an (effectively computable) arithmetic projection.

Proof. Similar to the proof of Theorem 4, using Theorem 3 instead of Corollary 1.

To sum up, given a system specification $\mathcal{S} = (\Sigma, \mathbf{v}, \Phi, \iota, \tau)$ and a safety problem $\upsilon(\mathbf{v})$, if the formulae $\Phi, \iota, \tau, \upsilon$ satisfy suitable syntactic restrictions so that our quantifier elimination results apply, *we can compute the arithmetic projection $\mathcal{S}_0$ of $\mathcal{S}$ and try to show that $\mathcal{S}_0$ is safe with respect to $\upsilon_0(\mathbf{v}_0)$* (the latter is the formula obtained in its turn by eliminating the higher order variables from $\upsilon(\mathbf{v})$).[11] Thus a model ckecking problem formulated in higher order logic can be solved via a model checking problem for counter systems (i.e. for system specifications in a purely arithmetic signature). The literature on distributed systems confirms that this is a viable approach: since long time it has been observed that counter systems [10,11,13] can be sufficient to specify problems like cache coherence or broadcast protocols. Recently, counter abstractions have been

[10] In view of condition (iii) of Proposition 1, we need also the observation that formulæ like (16) are closed under conjunctions.

[11] A more sophisticated strategy would preprocess the system specification $\mathcal{S}$ by artificially adding to it some extra integer variables counting certain definable sets (see the Appendix of [16] for an example on how this works).

effectively used also in the verification of fault-tolerant distributed algorithms [2, 22–24].

It should be noticed that safety problems for counter systems are themselves undecidable, however the sophisticated machinery (predicate abstraction [14], IC3 [8,20], etc.) developed inside the SMT community lead to impressively performing tools like μZ [21], nuXmv [7], SeaHorn [19], ... which are nowadays being successfully used to solve many verification problems regarding counter systems.

Arithmetic projections obtained by our methods are far from trivial: the reader may realize this by looking at the detailed analysis of a classical benchmark in the Appendix of [16] (more examples are described in [15]). In all such cases, the resulting safety model-checking problems for the arithmetic projections are solved instantaneously by μZ (the SMT-HORN module of z3).

5 Conclusions

We have investigated quantifier elimination results for fragments of higher order logic suggested by verification applications in the distributed algorithms area. We have shown how to apply such results in order to automatically produce arithmetic projections that can be effectively handled by state-of-the-art SMT-based model checkers. Similar applications can be devised for forward/backward model checking, along the lines sketched in [4]. We won't discuss and compare our approach here with the different approaches from the literature, the reader is referred to the final section of [15] for some information in this sense. We only point out that the main merit of the approach we propose is that of being purely *declarative*: our starting point is the informal description of the algorithms (e.g. in some pseudo-code) and our first step is a direct translation into a standard logical formalism (typically, classical Church type theory), without relying for instance on ad hoc automata devices or on ad hoc specification formalisms. We believe that this choice can ensure flexibility and portability of our methods.

A potentially weak point to be taken care is the *complexity* of the algorithms we employ: in fact, the procedure for quantifier elimination used in the proof of Theorems 1, 2, 3 produces super-exponential blow-ups of the size of the formulæ it is applied to. Notice however that, when building a counters simulation of a concrete algorithm, such a heavy procedure is applied to each instruction (or to each block of instructions) separately, i.e. not to the whole code. Moreover, it is not difficult to realize (going through the details for our benchmarks) that it is hardly the case that the quantifier elimination procedure is applied in its full generality: in fact, it is always applied to easier fragments, where complexity reduces (recall for instance the content of footnote 1). The same observation applies also to the instances of the Presburger quantifier elimination procedure that are invoked in our manipulations: usually, they are confined to difference logic formulæ or to formulæ where quantifiers can be eliminated by simple instantiations. The identification of such shortcuts and the study of the related

complexities is important for future work and preliminary to any substantial implementation effort.

Another delicate point is related to the *syntactic limitations* we require on the formulæ describing system specifications (see the statements of Theorems 4 and 5): such syntactic limitations are needed to ensure higher order quantifier elimination. Although it seems that a significant amount of benchmarks are captured despite such limitations, it is essential to develop techniques applicable in more general cases. To this aim, we observe that just overapproximations are needed to build simulations and that, even if the best simulation may not exist, still practically useful simulations might be produced. In fact, quantifier elimination is just an extreme solution to symbol elimination problems. Symbol elimination and interpolation are a well-known technique to build invariants, abstractions and overapproximations, and for this reason their investigation has deserved considerable attention in the automated reasoning literature [25]; extensions to higher-order fragments might be useful in our context too.

References

1. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy Abstraction with Interpolants for Arrays. In: LPAR. pp. 46–61 (2012)
2. Alberti, F., Ghilardi, S., Orsini, A., Pagani, E.: Counter Abstractions in Model Checking of Distributed Broadcast Algorithms: Some Case Studies. In: Proc. CILC. pp. 102–117. CEUR Proceedings (2016)
3. Alberti, F., Ghilardi, S., Pagani, E.: Counting Constraints in Flat Array Fragments. In: Proc. IJCAR. Lecture Notes in Computer Science, vol. 9706, pp. 65–81 (2016)
4. Alberti, F., Ghilardi, S., Pagani, E.: Cardinality Constraints for Arrays (decidability results and applications). Formal Methods in System Design (2017)
5. Andrews, P.B.: An introduction to mathematical logic and type theory: to truth through proof, Applied Logic Series, vol. 27. Kluwer Academic Publishers, Dordrecht, second edn. (2002)
6. Bjørner, N., von Gleissenthall, K., Rybalchenko, A.: Cardinalities and Universal Quantifiers for Verifying Parameterized Systems. In: Proc. of the 37th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI) (2016)
7. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: CAV. pp. 334–342 (2014)
8. Cimatti, A., Griggio, A.: Software model checking via IC3. In: CAV. pp. 277–293 (2012)
9. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Cubicle: A parallel smt-based model checker for parameterized systems - tool paper. In: CAV. pp. 718–724 (2012)
10. Delzanno, G.: Constraint-Based Verification of Parameterized Cache Coherence Protocols. Formal Methods in System Design 23(3), 257–301 (2003)
11. Delzanno, G., Esparza, J., Podelski, A.: Constraint-Based Analysis of Broadcast Protocols. In: Proc. of CSL. LNCS, vol. 1683, pp. 50–66 (1999)
12. Dragoj, C., Henzinger, T., Veith, H., Widder, J., Zufferey, D.: A Logic-based Framework for Verifying Consensus Algorithms. In: Proc. of VMCAI (2014)

13. Esparza, J., Finkel, A., Mayr, R.: On the Verification of Broadcast Protocols. In: Proc. of LICS. pp. 352–359. IEEE Computer Society (1999)
14. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL. pp. 191–202 (2002)
15. Ghilardi, S., Pagani, E.: Counter Simulations via Higher Order Quantifier Elimination: a preliminary report. In: Proc. of PxTP. EPTCS (2017), (preliminary workshop version available from authors' webpages)
16. Ghilardi, S., Pagani, E.: Second Order Quantifier Elimination: Towards Verification Applications. (2017), (extended version available from authors' webpages)
17. Ghilardi, S., Ranise, S.: Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis. Logical Methods in Computer Science 6(4) (2010)
18. Ghilardi, S., Ranise, S.: MCMT: A Model Checker Modulo Theories. In: IJCAR. pp. 22–29 (2010)
19. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: CAV. pp. 343–361 (2015)
20. Hoder, K., Bjørner, N.: Generalized Property Directed Reachability. In: SAT. pp. 157–171 (2012)
21. Hoder, K., Bjørner, N., deMoura, L.: μZ– An Efficient Engine for Fixed Points with Constraints. In: CAV. pp. 457–462 (2011)
22. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: Proc. Int'l Conf. on Formal Methods in Computer-Aided Design (FMCAD). pp. 201–209 (Aug 2013)
23. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms. In: Proc. Int'l SPIN Symposium on Model Checking of Software. Lecture Notes in Computer Science, vol. 7976, pp. 209–226. Springer (Jul 2013)
24. Konnov, I.V., Veith, H., Widder, J.: What You Always Wanted to Know About Model Checking of Fault-Tolerant Distributed Algorithms. In: Perspectives of System Informatics - 10th International Andrei Ershov Informatics Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24-27, 2015, Revised Selected Papers. pp. 6–21 (2015)
25. Kovács, L., Voronkov, A.: Interpolation and Symbol Elimination. In: Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings. pp. 199–213 (2009)
26. Kuncak, V., Nguyen, H.H., Rinard, M.: Deciding Boolean Algebra with Presburger Arithmetic. Journal of Automated Reasoning 36(3) (2006)
27. Lambek, J., Scott, P.J.: Introduction to higher order categorical logic, Cambridge Studies in Advanced Mathematics, vol. 7. Cambridge University Press, Cambridge (1988), reprint of the 1986 original
28. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. Warszawa (1929)
29. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.W.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Proc. CAV. pp. 198–216 (2015)
30. Schweikardt, N.: Arithmetic, First-Order Logic, and Counting Quantifiers. ACM TOCL pp. 1–35 (2004)