# Second Order Quantifier Elimination: Towards Verification Applications

Silvio Ghilardi, Elena Pagani

Università degli Studi di Milano, Milano, Italy

**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, 16] 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, 17]), 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.

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; however, we supply the detailed analysis of a benchmark in Appendix A (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). In Appendix B, we report for completeness the proof of the results we use taken from [4].

## 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 $\Omega$ 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' $\{\mathtt{a}_0, \dots, \mathtt{a}_\mathtt{k}\}$, where the $\mathtt{a}_\mathtt{i}$'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* $\Sigma$ 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 $\Sigma$-structures, called the *models* of $\Sigma$.[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 [25].

[2] Some notation we use might look slightly non-standard; it is similar to the notation of [26].

[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 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

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

Our constrained signatures always include the sort $\Omega$ of truth-values; terms of type $\Omega$ are called *formulae* (we use greek letters $\alpha, \beta, \dots, \phi, \psi, \dots$ 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, $\Sigma$ 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 $\phi$ 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. [26]). We shall use the roman letters $x, y, \dots, i, j, \dots, v, w, \dots$ 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 $\Omega$). 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 *$\Sigma$-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 $\Sigma$-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* $\phi$); $\phi$ 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 $\phi$ is valid and $\phi \models_\Sigma \psi$ (or just $\phi \models \psi$) to mean that $\phi \to \psi$ is valid; we say that $\phi$ and $\psi$ are *$\Sigma$-equivalent* (or just equivalent) iff $\phi \leftrightarrow \psi$ is valid.

### 2.1 Flat Cardinality Constraints

Let us fix a constrained signature $\Sigma$ for the remaining part of the paper. Such $\Sigma$ should be adequate for modeling parameterized systems, hence we assume that $\Sigma$ 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 $\Omega$, with the constants $\mathtt{tt}, \mathtt{ff}$ and the Boolean operations on them;
(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);

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 [28]).

(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 $\Sigma$ *finitary*; the subsignature $\Sigma_0$ of $\Sigma$ obtained by restricting to sorts and operations in (i)-(ii) is called the *arithmetic* subsignature of $\Sigma$.

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 $\Sigma$ 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 `Proc` $\to$ `Data` or of type `Proc` $\to \mathbb{Z}$ and a matrix-id is a variable of type `Proc` $\to$ (`Proc` $\to$ `Data`) or of type `Proc` $\to$ (`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 $\lambda$-abstractors and no quantifiers).
- *1-Flat formulæ*: these are formulæ of the kind $\phi(\sharp\{x \mid \psi_1\} / z_1, \dots, \sharp\{x \mid \psi_n\} / z_n)$, where $\phi(z_1, \dots, z_n), \psi_1, \dots, \psi_n$ are open and $x$ is a variable of type `Proc`.
- Given an index variable $i$, a formula $\phi$ 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 $\phi$ and the only terms occurring in $\phi$ 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 $\forall i\ (a(i) = c \to b(i) = d)$ is the same as $\sharp\{i \mid a(i) = c \to b(i) = d\} = \mathtt{N}$,[5] and similarly $\exists i\ (a(i) = c)$ can be re-written as $\sharp\{i \mid a(i) = c\} > 0$.

[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 $\alpha$-conversions in the paper.

*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 $\Sigma$ 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 $\alpha$ is open and pure. Actually, using disjunctive normal forms, we can assume that $\alpha$ 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 [27], 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 $\alpha$ is $x \neq y_1 \wedge \cdots \wedge x \neq y_n$. 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). $\dashv$

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 $\alpha$) inside a pure formula $\phi$. We can first remove from $\alpha$ arithmetic and data atoms, as well as index atoms not containing $x$, by the following equivalence (let $A$ be the atom to be removed):

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

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 $\phi$ entails $\bigwedge_{s \neq s'} i_s \neq i_{s'}$ (otherwise we can force this by making $\phi$ 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. $\dashv$

It is now convenient to introduce a notation for open (not necessarily pure) formulæ (from now on we shall reserve the letters $\alpha, \beta, \dots$ 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 $\phi$ 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 $\phi$ does not contain matrix-ids, we can write it as

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

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 $\mathbf{a} = a_1, \dots, a_n$ and $\underline{k} = k_1, \dots, k_m$, then $\mathbf{a}(\underline{k})$ is the tuple

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

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

[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 [29].

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

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

or (with some abuse of notation) shortly as

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

where $\underline{\beta}$ is a tuple of formulæ (we use the convention that $\sharp\{x \mid \underline{\beta}\}$ stands for the tuple of terms $\sharp\{x \mid \beta_1\}, \dots, \sharp\{x \mid \beta_s\}$). 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 report the proof in the Appendix B for the sake of completeness):

**Theorem 1.** *Suppose that $\phi$ 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, $\psi$ 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 $\phi$ 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 to formalize benchmarks, see Appendix A for an example):

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

$$\exists \mathbf{a}\, \forall i\, \exists \underline{y}\, \phi \qquad (3)$$

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

*Proof.* Let `Data` be enumerated as $\{\mathtt{a_0}, \dots, \mathtt{a_k}\}$; let $\underline{z}$ be the arithmetic variables occurring freely in (3) and let $\underline{k} = k_1, \dots, k_n$ 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 `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]

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

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

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

where

- the formula $\texttt{Diff}(\underline{k})$ 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 $\texttt{Data}$ 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 $\phi'$ 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.

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 $\mathtt{a}_\mathtt{j}$ in our enumerated data)

$$\forall i.\;\; \texttt{Diff}(i, \underline{k}) \wedge a(i) = \mathtt{a}_\mathtt{j} \to \exists \underline{y}\, \phi''(\underline{z}, \underline{y}, i, \underline{k}, \sharp\{x \mid \underline{\beta}'(\underline{z}, \underline{y}, x, i, \underline{k}, a(x))\}) \tag{5}$$

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

First observe that $\phi''$ 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 $\texttt{Proc}$ (and there are no function symbols of sort $\texttt{Proc}$), the only atoms of sort $\texttt{Proc}$ 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 $\texttt{Diff}(i, \underline{k})$ in the antecedent of the implication of (5). As a consequence $\phi''$ *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 $\texttt{Data}$-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 $\phi''$ as

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

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

$$\forall i.\ \texttt{Diff}(i, \underline{k}) \wedge a(i) = \texttt{a}_\texttt{j} \to \exists \underline{y}\, \phi''(\underline{z}, \underline{y}, \underline{m} + \sharp\{x \mid \texttt{Diff}(x, i, \underline{k}) \wedge \underline{\beta}''(a(x))\}) \quad (7)$$

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

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

$$\forall i.\ \texttt{Diff}(i, \underline{k}) \wedge a(i) = \texttt{a}_\texttt{j} \to \exists \underline{y}\, \theta_j(\underline{y}, \underline{z}, \underline{u}) \quad (8)$$

where the formulæ $\theta_j$ 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

$$\forall i.\ \texttt{Diff}(i, \underline{k}) \wedge a(i) = \texttt{a}_\texttt{j} \to \theta'_j(\underline{z}, \underline{u}) \quad (9)$$

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

$$\exists i\ (\texttt{Diff}(i, \underline{k}) \wedge a(i) = \texttt{a}_\texttt{j}) \to \theta'_j(\underline{z}, \underline{u}) \quad (10)$$

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

and finally as

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

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 $\vartheta$ 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 Appendix A 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 $\phi$ are $i$-uniform and that all array-ids $\mathbf{a}$ occurring in $\phi$ 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 $\phi$ 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, $\psi$ 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 $\psi$. 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 work (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) $\Sigma$ is a constrained signature, (ii) $\mathbf{v}$ is a tuple of variables, (iii) $\Phi, \iota$ are $\mathbf{v}$-formulæ, (iv) $\tau$ 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, $\iota$ is meant to describe initial states and $\tau$ is meant to describe the transition relation. The $\mathbf{v}$-formula $\Phi$, 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* $\upsilon$ 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* $\Sigma_0$ of $\Sigma$ is a signature obtained from $\Sigma$ by dropping some symbols of $\Sigma$ and taking as $\Sigma_0$-models the class $\mathcal{C}_{\Sigma_0}$ of the restrictions $\mathcal{M}_{|\Sigma_0}$ to the $\Sigma_0$-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 $\Sigma_0$ is a subsignature of $\Sigma$ 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 $\upsilon_0$, so it is $\mathcal{S}$ with respect to $\upsilon$.*[9]

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 $\Sigma_0$ is the arithmetical subsignature of $\Sigma$ 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 $\Phi, \iota, \tau$ do not contain matrix-ids and are of the kind $\exists k_1 \cdots \exists k_n \phi$ for a 1-flat formula $\phi$ and for index variables $k_1, \dots, k_n$, then $\mathcal{S} = (\Sigma, \mathbf{v}, \Phi, \iota, \tau)$ has an (effectively computable) arithmetic projection.*

[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 $\mathbf{v}_0$).

*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 $k_1, \dots, k_n$ 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 $\Phi, \iota, \tau$ be disjunctions of formulæ of the kind*

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

*where $\phi$ 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 $\phi$; 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 effectively used also in the verification of fault-tolerant distributed algorithms [2, 21–23].

It should be noticed that safety problems for counter systems are themselves undecidable, however the sophisticated machinery (predicate abstraction [14], IC3 [8,19], etc.) developed inside the SMT community lead to impressively performing tools like $\mu Z$ [20], nuXmv [7], SeaHorn [18], ... which are nowadays being successfully used to solve many verification problems regarding counter systems.

[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 artificially add to $\mathcal{S}$ some additional integer variables counting some definable sets, see the Appendix A for an example on how this works.

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 Appendix A below (more examples are described in [15]). In all such cases, the resulting safety model-checking problems for the arithmetic projections are solved instantaneously by $\mu Z$ (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 computation 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 smaller 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 small fragments 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 [24]; extensions to higher-order fragments need to be investigated and might be useful in our context.

## References

1. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy Abstraction with Interpolants for Arrays. In *LPAR*, pages 46–61, 2012.
2. F. Alberti, S. Ghilardi, A. Orsini, and E. Pagani. Counter Abstractions in Model Checking of Distributed Broadcast Algorithms: Some Case Studies. In *Proc. CILC*, CEUR Proceedings, pages 102–117, 2016.
3. F. Alberti, S. Ghilardi, and E. Pagani. Counting Constraints in Flat Array Fragments. In *Proc. IJCAR*, volume 9706 of *Lecture Notes in Computer Science*, pages 65–81, 2016.
4. F. Alberti, S. Ghilardi, and E. Pagani. Cardinality Constraints for Arrays (decidability results and applications). *Formal Methods in System Design*, 2017.
5. Peter B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof*, volume 27 of *Applied Logic Series*. Kluwer Academic Publishers, Dordrecht, second edition, 2002.
6. N. Bjørner, K. von Gleissenthall, and A. Rybalchenko. 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. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv Symbolic Model Checker. In *CAV*, pages 334–342, 2014.
8. A. Cimatti and A. Griggio. Software model checking via IC3. In *CAV*, pages 277–293, 2012.
9. Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems - tool paper. In *CAV*, pages 718–724, 2012.
10. G. Delzanno. Constraint-Based Verification of Parameterized Cache Coherence Protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
11. G. Delzanno, J. Esparza, and A. Podelski. Constraint-Based Analysis of Broadcast Protocols. In *Proc. of CSL*, volume 1683 of *LNCS*, pages 50–66, 1999.
12. C. Dragoj, T. Henzinger, H. Veith, J. Widder, and D. Zufferey. A Logic-based Framework for Verifying Consensus Algorithms. In *Proc. of VMCAI*, 2014.
13. J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proc. of LICS*, pages 352–359. IEEE Computer Society, 1999.
14. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.
15. S. Ghilardi and E. Pagani. Counter Simulations via Higher Order Quantifier Elimination: a preliminary report. In *Proc. of PxTP*, EPTCS, 2017. Preliminary workshop version available from authors' webpages.

16. S. Ghilardi and S. Ranise. Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis. *Logical Methods in Computer Science*, 6(4), 2010.
17. S. Ghilardi and S. Ranise. MCMT: A Model Checker Modulo Theories. In *IJCAR*, pages 22–29, 2010.
18. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn Verification Framework. In *CAV*, pages 343–361, 2015.
19. K. Hoder and N. Bjørner. Generalized Property Directed Reachability. In *SAT*, pages 157–171, 2012.
20. K. Hoder, N. Bjørner, and L. deMoura. $\mu$Z– An Efficient Engine for Fixed Points with Constraints. In *CAV*, pages 457–462, 2011.
21. A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Proc. Int'l Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 201–209, Aug. 2013.
22. A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms. In *Proc. Int'l SPIN Symposium on Model Checking of Software*, volume 7976 of *Lecture Notes in Computer Science*, pages 209–226. Springer, Jul. 2013.
23. Igor V. Konnov, Helmut Veith, and Josef Widder. 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*, pages 6–21, 2015.
24. Laura Kovács and Andrei Voronkov. Interpolation and Symbol Elimination. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, pages 199–213, 2009.
25. Viktor Kuncak, Huu Hai Nguyen, and Martin Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *Journal of Automated Reasoning*, 36(3), 2006.
26. J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1988. Reprint of the 1986 original.
27. M. Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt*. Warszawa, 1929.
28. Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *Proc. CAV*, pages 198–216, 2015.
29. N. Schweikardt. Arithmetic, First-Order Logic, and Counting Quantifiers. *ACM TOCL*, pages 1–35, 2004.
30. T.K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

**Algorithm 1** Algorithm for broadcasting and accepting $(p, m, k)$:

```
Round $k$:
    Phase $2k-1$: process $p$ sends $(init, p, m, k)$ to all processes;
    Phase $2k$: each process executes the following for each $m \in M$:
      if received $(init, p, m, k)$ from $p$ in phase $2k-1$
          then send $(echo, p, m, k)$ to all;
      if received $(echo, p, m, k)$ from at least $\mathtt{N} - t$ distinct processes in phase $2k$
          then accept $(p, m, k)$;

Round $r \geq k+1$:
    Phase $2r-1, 2r$: each process executes the following:
      if received $(echo, p, m, k)$ from at least $\mathtt{N} - 2t$ distinct processes
          in previous phases and not sent $(echo, p, m, k)$
          then send $(echo, p, m, k)$ to all;
      if received $(echo, p, m, k)$ from at least $\mathtt{N} - t$ distinct processes
          in this and previous phases, then accept $(p, m, k)$.
```

## A A Detailed Example

System specifications matching the syntactic restrictions of Theorems 4,5 are powerful enough to accurately formalize many relevant examples; we supply in this section a detailed analysis of Byzantine Broadcast Primitive protocol.

Byzantine Broadcast Primitive (BBP) [30] is summarized in Algorithm 1. The algorithm is supposed to work correctly for `N` processes, of which at most $t < \mathtt{N}/3$ are faulty. In Figure 1 we reported literally the pseudo-code from the original paper [30] and in Table 1 we reported the correctness, unforgeability and relay properties to be model-checked (still taken from [30]).

As it often happens with the literature in distributed systems, high-level specifications are insufficient for formal verification, and filling missing details is a rather challenging task because of various ambiguities and hidden assumptions one needs to take care of. The algorithm in Figure 1 is specified in a *synchronous* way, thus we have a round counter $r$ (which is an integer variable); a round terminates when all relevant instructions are completed by the correct processes (faulty processes may behave arbitrarily). For our system specification, we use the following variables besides $r$ (the sort `Data` is the sort of truth values, i.e. our array-ids and matrix-ids are Boolean valued):

- the array-ids $SE, AC$: here $SE(x)$ says whether the process $x$ sent-or-is-sending $(echo, p, m, k)$ to all, $AC(x)$ says whether $x$ accepted or not;
- the array-id $C$ expresses that $x$ is a correct process (this array-id is immutable, i.e, it is not modified along the execution of the algorithm);
- the array-id $RI$ says whether $x$ has received $(init, p, m, k)$ from $p$ (this is also immutable);
- the matrix-id $R(x, y)$ says that $x$ has received $(echo, p, m, k)$ from $y$.

|  |
|---|
| **Correctness:** If correct process $p$ broadcasts $(p, m, k)$ in round $k$, then every correct process accepts $(p, m, k)$ in the same round. |
| **Unforgeability:** If a process $p$ is correct and does not broadcast $(p, m, k)$, then no correct process ever accepts $(p, m, k)$. |
| **Relay:** If a correct process accepts $(p, m, k)$ in round $r > k$, then every other correct process accepts $(p, m, k)$ in round $r + 1$ or earlier. |

**Table 1:** Properties to be certified.

The transition formula $\tau$ is the conjunction of two formulæ $\tau_1$ and $\tau_2$, relative to round $k$ (phase $2k$)[12] and to rounds $r \geq k$, respectively. Below we use abbreviations like $\forall i \in C\ \cdots$ for $\forall i\ (C(i) \to \cdots$; when specifying $\tau_1, \tau_2$ we leave implicit the condition saying that $C, RI$ are immutable (strictly speaking, we should use primed variables $C', RI'$ and write explicitly $\forall x(C(x) \leftrightarrow C'(x))$ and $\forall x(RI(x) \leftrightarrow RI'(x))$). Let us first introduce $\tau_1$:[13]

$$\begin{aligned} & r = k\ \wedge\ r' = k+1\ \wedge\ \forall i \in C\ (SE'(i) \leftrightarrow RI(i))\ \wedge \\ & \wedge\ \forall i \in C\, \forall x \in C\ (R'(i,x)\ \leftrightarrow SE'(x))\ \wedge \\ & \wedge\ \forall i \in C\ (AC'(i) \leftrightarrow \sharp\{x \mid R'(i,x)\} \geq \mathtt{N} - t)\ \ . \end{aligned} \tag{18}$$

Notice that the above formula (18) fits the constraints of Theorem 5: to see this, observe for instance that $\forall i \in C\, \forall x \in C\ (R'(i,x)\ \leftrightarrow SE'(x))$ can be written as $\forall i \in C\ \mathtt{N} = \sharp\{x \mid C(x) \to (R'(i,x)\ \leftrightarrow SE'(x))\}$, where $R'$ is uniform with respect to $i$ (we won't repeat similar observations below).

We now specify $\tau_2$:

$$\begin{aligned} & r > k\ \wedge\ r' = r+1\ \wedge\ \forall i\, \forall x\ (R(i,x) \to R'(i,x))\ \wedge \\ & \wedge\ \forall i \in C\ [SE'(i) \leftrightarrow SE(i) \vee \sharp\{x \mid R(i,x)\} \geq N - 2t]\ \wedge \\ & \wedge\ \forall i \in C\, \forall x \in C\ [R'(i,x)\ \leftrightarrow R(i,x) \vee (\neg SE(x) \wedge SE'(x))]\ \wedge \\ & \wedge\ \forall i \in C\ [AC'(i) \leftrightarrow (AC(i) \vee \sharp\{x \mid R'(i,x)\} \geq \mathtt{N} - t)]\ \ . \end{aligned} \tag{19}$$

The initialization formula $\iota$ expresses the situation after round $k$ , phase $2k-1$: it does not involve any constraint on $RI$ because some correct processes may have received $(init, p, m, k)$ and some other may not. If the initiator process $p$ is correct, all correct processes should have received this message (we shall distinguish the case where $p$ is correct from the case in which it is not when specifying the properties to be model-checked). Thus our $\iota$ is

$$\begin{aligned} & \forall i\ (\neg SE(i) \wedge \neg AC(i))\ \wedge \forall i\, \forall x\ \neg R(i,x)\ \wedge \\ & \wedge\ r = k\ \wedge\ 3t < \mathtt{N}\ \wedge\ t \geq \sharp\{x \mid \neg C(x)\} \end{aligned} \tag{20}$$

[12]Phase $2k-1$ will be taken care by the initialization formula.

[13]When specifying $\tau_1, \tau_2$, we desume from Figure 1 that if a message is sent to a correct process by a correct process, then the message is delivered inside the round.

(we inserted into $\iota$ also our constraints on $\mathtt{N}, t$).

To accomplish our plan, we need the following steps:

(i) complete our system specification by adding further variables counting the cardinality of some definable sets (these counters should be accurately chosen, so as to be sufficient to express the properties we want to check, they will be the variables for our projected simulation);
(ii) complete our system specification by supplying an invariant $\Phi$ satisfying the conditions of Definition 1 (this step is not needed in principle - $\Phi$ might be $\top$ - but an invariant can considerably help successive steps);
(iii) compute the projected system by eliminating the higher-order variables, according to the procedure of the proof of Theorem 5;
(iv) formalize and check the desired properties by using an SMT-based infinite state model-checker.

Operations (i)-(ii) require manual intervention (but automatic support can be used e.g. to check whether a proposed invariant is really an invariant); by contrast, (iii)-(iv) are mechanical.

**As for (i)**, we decide to introduce four counters, namely for the set of faulty processes and for the set of correct processes who received init, sent-or-are-sending echo and accepted. That is, we add to our system specification the integer variables $f, z_{RI}, z_{SE}, z_{AC}$; moreover, the formulæ $\iota, \tau_1, \tau_2$ are modified as follows

$$\iota^+ :\equiv \ \iota \ \wedge \ \delta, \qquad \tau_1^+ :\equiv \ \tau_1 \ \wedge \ \delta \ \wedge \delta', \qquad \tau_2^+ :\equiv \ \tau_2 \ \wedge \ \delta \ \wedge \delta' \tag{21}$$

where the auxiliary formulæ $\delta, \delta'$ are supplied in Figure 1 (notice that $f, z_{RI}$ need not occur primed in $\delta'$ because they are immutable like $C, RI$).

**As for (ii)**, we propose the invariant $\Phi$ given by

$$3t < \mathtt{N} \wedge f \leq t \wedge \delta \wedge \forall i \in C\ \forall x \in C\ (SE(x) \leftrightarrow R(i, x)) \tag{22}$$

The last conjunct says that the sending actions in our algorithm are broadcast actions, so that a sending correct process (in any round) sends the message to all processes and *all correct processes* get it. This invariant *is checked automatically*: the formulæ (15) expressing that (22) is an invariant can be proved to hold by a trivial instantiation (consisting in instantiating $i, x$ as themselves) followed by Boolean reasoning (this strategy is incomplete in the general case, but is sufficient here).

We now have completed our system specification

$$\mathcal{S} = (\Sigma, \{C, RI, SE, AC, R, f, z_{RI}, z_{SE}, z_{AC}\}, \Phi, \iota^+, \tau_1^+ \vee \tau_2^+)$$

**For (iii)**, we need to compute the projections over the integer variables $f, z_{RI}, z_{SE}, z_{AC}$. This requires a complex procedure (following the proof of Theorem 5), yieldying to the system specification

$$\tilde{\mathcal{S}} = (\tilde{\Sigma}, \{f, z_{RI}, z_{SE}, z_{AC}\}, \tilde{\Phi}, \tilde{\iota}, \tilde{\tau}_1 \vee \tilde{\tau}_2)$$

$$\begin{array}{ll}
\delta : \equiv & z_{SE} = \sharp\{x \in C \mid SE(x)\} \wedge z_{AC} = \sharp\{x \in C \mid AC(x)\} \wedge \\
 & \wedge z_{RI} = \sharp\{x \in C \mid RI(x)\} \wedge f = \sharp\{x \mid \neg C(x)\} \\
\delta' : \equiv & z'_{SE} = \sharp\{x \in C \mid SE'(x)\} \ \wedge \ z'_{AC} = \sharp\{x \in C \mid AC'(x)\} \\
\psi : \equiv & 0 \leq 3t < \mathtt{N} \ \wedge \ 0 \leq f \leq t \ \wedge \ 0 \leq z_{SE} \leq \mathtt{N} - f \ \wedge \\
 & 0 \leq z_{AC} \leq \mathtt{N} - f \ \wedge \ 0 \leq z_{RI} \leq \mathtt{N} - f \\
\psi' : \equiv & 0 \leq z_{SE'} \leq \mathtt{N} - f \ \wedge \ 0 \leq z_{AC'} \leq \mathtt{N} - f \\
p_1 : \equiv & z_{SE} + f \geq \mathtt{N} - 2t \\
p_2 : \equiv & z_{SE'} + f \geq \mathtt{N} - t \\
p_3 : \equiv & z_{SE} \geq \mathtt{N} - 2t \\
p_4 : \equiv & z_{SE'} \geq \mathtt{N} - t \\
p_5 : \equiv & p_1 \wedge z_{SE'} - z_{SE} < t \wedge \neg p_4
\end{array}$$

**Fig. 1:** Auxiliary formulæ.

where $\tilde{\Sigma}$ is the signature of Presburger arithmetic (enriched with free constants $\mathtt{N}, t$) and $\tilde{\Phi}, \tilde{\iota}_c, \tilde{\iota}_u, \tilde{\tau}_1, \tilde{\tau}_2$ are described below (we use the auxiliary formulæ $\psi, \psi', p_1 - p_5$ introduced in Figure 1). First, $\tilde{\Phi}$ is $\psi$ and $\tilde{\iota}$ is $\psi \ \wedge \ z_{SE} = 0 \ \wedge z_{AC} = 0 \wedge \ r = k$; moreover $\tilde{\tau}_1$ is

$$\begin{gathered}
\psi \ \wedge \psi' \wedge \ r = k \ \wedge \ r' = k + 1 \ \wedge \ z_{SE'} = z_{RI} \ \wedge \\
\wedge \ (z_{RI} \geq \mathtt{N} - t \to z_{AC'} = \mathtt{N} - f) \ \wedge \ (z_{RI} + f < \mathtt{N} - t \to z_{AC'} = 0)
\end{gathered}$$

and $\tilde{\tau}_2$ is $\psi \wedge \psi' \wedge \ r > k \ \wedge \ r' = r + 1 \wedge \Theta$, where $\Theta$ is the disjunction of the following 6 formulæ:

$$\begin{array}{l}
p_3 \wedge p_4 \wedge z_{SE'} = \mathtt{N} - f \wedge z_{AC'} = \mathtt{N} - f \\
\neg p_3 \wedge p_4 \wedge p_1 \wedge z_{AC'} = \mathtt{N} - f \wedge z_{SE} \leq z_{SE'} \\
\neg p_3 \wedge \neg p_4 \wedge p_1 \wedge p_2 \wedge p_5 \wedge z_{AC} \leq z_{AC'} \wedge z_{SE} \leq z_{SE'} \\
\neg p_3 \wedge \neg p_4 \wedge p_1 \wedge p_2 \wedge \neg p_5 \wedge z_{AC} \leq z_{AC'} \wedge z_{SE} \leq z_{SE'} \wedge z_{SE'} - z_{SE} \leq z_{AC'} \\
\neg p_3 \wedge \neg p_4 \wedge p_1 \wedge \neg p_2 \wedge z_{AC} = z_{AC'} \wedge z_{SE} \leq z_{SE'} \\
\neg p_3 \wedge \neg p_4 \wedge \neg p_1 \wedge \neg p_2 \wedge z_{AC} = z_{AC'} \wedge z_{SE} = z_{SE'}
\end{array}$$

**As for (iv)**, we need to express the safety properties to be model-checked in higher order logic and then eliminate second order variables from them. In our case however, it is possible and more simple to express safety properties directly in terms of our projection counters. The *correctness* property to be checked for $\mathcal{S}$ is that the system never reaches a status satisfying $z_{RI} = \mathtt{N} - f \wedge r > k \wedge z_{AC} \neq \mathtt{N} - f$; the unforgeability property to be checked for $\mathcal{S}$ is that the system never reaches a status with $z_{RI} = 0 \wedge z_{AC} > 0$. Both problems are instantaneously solved by SMT-based tools.

Proving the relay property is more involved. This property says that *in every reachable state*, the 'relay property' is true; since we cannot describe the set of reachable states using our formulæ, the idea is *to split the relay property into two safety properties.*[14]

[14]This strategy indeed follows the informal proof given in [30].

Our first safety problem is to check that $\mathcal{S}$ never reaches a status satisfying $z_{AC} > 0 \wedge z_{SE} < \mathtt{N} - 2t$ or a status satisfying $z_{AC} > 0 \wedge r \leq k$. We then change the initialization formula $\iota$ of $\mathcal{S}$ into $z_{AC} > 0 \wedge z_{SE} \geq \mathtt{N} - 2t \wedge r = \mathtt{r}_0 \wedge \mathtt{r}_0 > k$ (here $\mathtt{r}_0$ is a new free constant) and then we solve our second safety problem, namely that the system so modified cannot reach a state satisfying $r = \mathtt{r}_0 + 1 \wedge z_{AC} \neq \mathtt{N} - f$. These two problems are also solved instantaneously by SMT-based tools.[15]

**Remarks and Comments.**

The six disjuncts of our final $\Theta$ supply a fine analysis of the algorithm: we underline that this (slightly involved) analysis is *produced by our automatic procedure*. To interpret it, recall that $z_{AC'}$ (resp. $z_{AC}$) counts the correct processes that accept in the current or in previous rounds (resp. in previous rounds) and that $z_{SE'}$ (resp. $z_{SE}$) counts the correct processes that send echo in the current or in previous rounds (resp. in previous rounds). The first disjunct analyzes what happens when $z_{SE} \geq \mathtt{N} - 2t$ and $z_{SE'} \geq \mathtt{N} - t$ both hold: here all correct processes send (or already sent) echo and accept (or already accepted). The second disjunct covers the situation where we have $z_{SE} < \mathtt{N} - 2t$ and $z_{SE'} \geq \mathtt{N} - t$: in this case all correct process accept or already-accepted because they received enough echoes (if $t = f$, then all correct processes also send-or-already-sent echo, in the general case we only have $z_{SE'} \geq z_{SE}$ and $z_{SE'} \geq \mathtt{N} - t$). The third and the fourth disjunct cover the case where we have $z_{SE} < \mathtt{N} - 2t$ and $z_{SE'} < \mathtt{N} - t$, but $z_{SE} + f \geq \mathtt{N} - 2t$ and $z_{SE'} + f \geq \mathtt{N} - t$: here both the numbers of processes who accept-or-accepted and send-or-sent echo may non-deterministically increase (depending on the number of faulty echoes each correct process receives), but if $z_{SE'} - z_{SE} \geq t$ (this is the case of the fourth disjunct), then all processes that send echo in the round (and not before) got enough echoes from faulty and correct processes so as also to accept[16] and we have $z_{SE'} - z_{SE} \leq z_{AC'}$. The fifth disjunct analyzes the situation where the number of accepting correct processes does not increase, but the number of correct processes who sent-or-send echo is possibly increasing. The last disjunct is relative to a situation where the status of correct processes does not change.

The system $\tilde{\mathcal{S}}$ is not bisimilar to the system $\mathcal{S}$: in fact, there are two main differences among them, showing that some information has been indeed abstracted out when passing from $\mathcal{S}$ to its projection $\tilde{\mathcal{S}}$. First, in $\tilde{\mathcal{S}}$ the behavior of byzantine processes is not explicitly modeled (only their effect on correct processes is modeled); second, the number of messages sent by byzantine processes is not recorded when passing from a round to the next one (in a sense, messages sent by byzantine processes can be freely 'added or withdrawn'). Both abstractions

[15] Executable files (in the extended SMT-LIB format of constrained Horn clauses) for $\mu Z$ containing all problems discussed here (and more) are available at `http://users.mat.unimi.it/users/ghilardi/counters_simulations`.

[16] In fact, they got enough echoes in the first phase of the round to reach the threshold $\mathtt{N} - 2t$ and receive further $z_{SE'} - z_{SE} \geq t$ echoes in the second phase, so that they also reach the threshold $\mathtt{N} - t$.

are not new, see [21] and [2]; what is different here is that such abstractions are not due to a manual design choice, but they come out automatically from our quantifier elimination procedures.

We underline that the BBP algorithm has different (e.g. asynchronous, not round-based) folklore versions, but our method applies also to these alternative versions. In this section, we tried to analyze the algorithm following strictly its informal specification from the literature; if coarser formalizations are employed, weaker quantifier elimination results are sufficient, like in [15], where the arithmetic projections (and the related $\mu Z$ files) are obtained not manually, but by running a computer program.

## B Proof of Theorem 1

**Lemma 3.** *If $\phi$ is an open formula and $\underline{d}'$ are arithmetic and data variables, then $\exists \underline{d}'\, \phi$ is equivalent to an open formula.*

*Proof.* Let $\phi$ be $\alpha(\underline{z}, \underline{k}, \mathbf{a}(\underline{k}), \underline{d})$ and let $\underline{d}'$ be data variables.[17] Suppose that $\underline{d} = \underline{d}'\underline{d}''$; then $\exists \underline{d}' \alpha(\underline{z}, \underline{k}, \underline{e}, \underline{d})$, by Lemma (1), is equivalent to a pure formula $\beta(\underline{z}, \underline{k}, \underline{e}, \underline{d}'')$, so that, applying a substitution, $\exists \underline{d}' \alpha(\underline{z}, \underline{k}, \mathbf{a}(\underline{k})/\underline{e}, \underline{d})$ is equivalent to $\beta(\underline{z}, \underline{k}, \mathbf{a}(\underline{k})/\underline{e}, \underline{d}'')$, which is as desired. The case in which $\underline{d}$ are arithmetic variables is treated similarly. $\dashv$

**Theorem 1**. *Suppose that $\phi$ 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, $\psi$ is 1-flat and contains only the array-ids $\mathbf{a}$.*

*Proof.* We start with a formula having the form

$$\exists \mathbf{a}' \quad \alpha(\underline{z}, \underline{k}, \mathbf{a}(\underline{k}), \mathbf{a}'(\underline{k}), \underline{d}, \sharp\{x \mid \underline{\beta}(\underline{z}, x, \underline{k}, \mathbf{a}(x), \mathbf{a}'(x), \mathbf{a}(\underline{k}), \mathbf{a}'(\underline{k}), \underline{d})\}) \tag{23}$$

where $\underline{k}$ are index variables, $\underline{z}$ arithmetic variables and $\underline{d}$ data variables. We can first get rid of the terms $\mathbf{a}(\underline{k}), \mathbf{a}'(\underline{k})$, as follows. Suppose that $\mathbf{a} = a_1, \dots, a_n$, $\mathbf{a}' = a'_1, \dots, a'_{n'}$ and $\underline{k} = k_1, \dots, k_m$; we introduce new variables

$$\underline{e} = e_{11}, \dots, e_{nm}, e'_{11}, \dots, e'_{n'm}$$

and rewrite (2) as

$$\exists \underline{e}.\ \bigwedge_{i,j} a_i(k_j) = e_{ij} \wedge \exists \mathbf{a}' \begin{pmatrix} \bigwedge_{i,j} \sharp\{x \mid x = k_i \wedge a'_j(x) = e'_{ij}\} = 1 \wedge \\ \wedge \ \alpha(\underline{z}, \underline{k}, \underline{e}, \underline{d}, \sharp\{x \mid \underline{\beta}(\underline{z}, x, \underline{k}, \mathbf{a}(x), \mathbf{a}'(x), \underline{e}, \underline{d})\}) \end{pmatrix}$$

[17] In (1), there are no matrix-ids; if there were also matrix-ids, then the argument would be the same (we do not insist, because we shall need the lemma only for formulæ without matrix-ids).

Thus it will be sufficient to eliminate the $\exists \mathbf{a}'$ from formulæ of the kind

$$\exists \mathbf{a}' \ \gamma_0(\underline{z}, \underline{k}, \underline{d}', \sharp\{x \mid \underline{\beta}(\underline{z}, x, \underline{k}, \mathbf{a}(x), \mathbf{a}'(x), \underline{d}')\}) \tag{24}$$

(here the $\underline{d}'$ include the old $\underline{d}$ and the $\underline{e}$ - the latter are existentially quantified and will remain such in the final outcome).

If we suppose that $\underline{\beta}$ is $\beta_1, \ldots, \beta_t$, we can set $K := \wp(\{1, \ldots, t\})$ and introduce for every $r \in K$ a new existentially quantified arithmetic variable $u_r$, thus rewriting (24) as

$$\exists \underline{u}, \mathbf{a}'. \ \bigwedge_r u_r = \sharp\{x \mid \beta_r(\underline{z}, x, \underline{k}, \mathbf{a}(x), \mathbf{a}'(x), \underline{d}')\} \wedge \gamma(\underline{z}, \underline{k}, \underline{d}', \underline{u}) \tag{25}$$

where $\underline{u}$ is the tuple formed by the $u_r$'s (varying $r$) and $\beta_r$ is the 'Venn region' $\bigwedge_{l \in r} \beta_l \wedge \bigwedge_{l \not\in r} \neg\beta_l$; the formula $\gamma$ is obtained from $\gamma_0$ by replacing, for all $l$, the term $\sharp\{x \mid \beta_l\}$ with $\sum_{l \in r} u_r$. Notice that at this point $\gamma$ is pure and the new $\beta_r$'s are a partition (i.e. they are mutually inconsistent and $\bigvee_r \beta_r$ is valid).

To continue, following the technique in [3], we need a further 'Venn region decomposition' $\delta_S$. For every $S \in \wp(K)$ let $\delta_S(\underline{z}, x, \underline{k}, \mathbf{a}(x), \underline{d}')$ be the pure formula

$$\bigwedge_{r \in S} \exists y \, \beta_r(\underline{z}, x, \underline{k}, \mathbf{a}(x), y, \underline{d}') \wedge \bigwedge_{r \not\in S} \neg \exists y \, \beta_r(\underline{z}, x, \underline{k}, \mathbf{a}(x), y, \underline{d}')$$

(here data quantifiers $\exists y$ can be eliminated using Lemma 3). We claim that the formula (25) is equivalent to the formula obtained by prefixing the existential quantifiers $\exists u_r$ (varying $r \in K$), $\exists u_S$ (varying $S \in \wp(K)$) and $\exists_{r,S}$ (varying $S \in \wp(K)$ and $r \in K$) to the formula

$$\begin{aligned} &\bigwedge_{S \in \wp(K)} u_S = \sharp\{x \mid \delta_S(\underline{z}, x, \underline{k}, \mathbf{a}(x), \underline{d}')\} \wedge \bigwedge_{S \in \wp(K)} \left( u_S = \sum_{r \in S} u_{r,S} \right) \wedge \\ &\wedge \bigwedge_{r \in K} \left( u_r = \sum_{S \in \wp(K), r \in S} u_{r,S} \right) \wedge \bigwedge_{r \in S \in \wp(K)} u_{r,S} \geq 0 \wedge \gamma(\underline{z}, \underline{k}, \underline{d}', \underline{u}) \end{aligned} \tag{26}$$

*Suppose that* (26) *is satisfied under the assignment* $\mathcal{I}$ to the free variables occurring in it (for simplicity, we use the same name for a free variable and for the integer assigned to it by $\mathcal{I}$). Let us assume that `Proc` is interpreted (up to a finite sets bijection) as the interval $[0, \mathtt{N})$; we need to define for all $i \in [0, N)$ the tuple $\mathbf{a}'(i)$ - namely $a'_s(i)$, for all $s = 1, \ldots, n'$. For every $r = 1, \ldots, K$ this must be done in such a way that there are exactly $u_r$ elements taken from $[0, N)$ satisfying $\beta_r(\underline{z}, x, \underline{k}, \mathbf{a}(x), \mathbf{a}'(x), \underline{d}')$. The interval $[0, \mathtt{N})$ can be partioned by associating with each $i \in [0, \mathtt{N})$ the set $i_S = \{r \in K \mid \exists \underline{y} \, \beta_r(\underline{z}, i, \underline{k}, \mathbf{a}(i), \underline{y}, \underline{d}')$ holds under $\mathcal{I}\}$. From the fact that (26) is true, we know that for every $S \in \wp(K)$ the number of the $i$'s such that $i_S = S$ is $u_S$; for every $r \in S$, pick $u_{r,S}$ among them and, for these selected $i$, let the $s$-tuple $\mathbf{a}'(i)$ be equal to an $s$-tuple $\underline{y}$ such that $\beta_r(\underline{z}, i, \underline{k}, \mathbf{a}(i), \underline{y}, \underline{d}')$ holds (for this tuple $\underline{y}$, since the $\beta_r$'s are a partition,

$\beta_h(\underline{z}, i, \underline{k}, \mathbf{a}(i), \underline{y}, \underline{d}')$ does not hold, if $h \neq r$). Since $u_S = \sum_{r \in S} u_{r,S}$ and since $\sum_S u_S$ is equal to $\mathtt{N}$ (because the formulæ $\bigwedge_{r \in S} \exists \underline{y}\, \beta_r \wedge \bigwedge_{r \notin S} \forall \underline{y} \neg \beta_r$ are a partition), the definition of the $\mathbf{a}'$ is complete. The formula (25) is true by construction.

On the other hand *suppose that* (25) *is satisfiable* under an assignment $\mathcal{I}$; we need to find $\mathcal{I}(u_S)$, $\mathcal{I}(u_{l,S})$ (we again indicate them simply as $u_S, u_{l,S}$) so that (26) is true (the $u_r$ are already given since (25) is true). For $u_S$ there is no choice, since $u_S = \sharp\{x \mid \delta_S(\underline{z}, x, \underline{k}, \mathbf{a}(x), \underline{d}')\}$ must hold; for $u_{r,S}$, we take it to be the cardinality of the set of the $i$ such that $\beta_r(\underline{z}, i, \underline{k}, \mathbf{a}(i), \mathbf{a}'(i), \underline{d}')$ holds under $\mathcal{I}$ and $S = \{h \in K \mid \exists \underline{y}\, \beta_h(\underline{z}, i, \underline{k}, \mathbf{a}(i), \underline{y}, \underline{d}')$ holds under $\mathcal{I}\}$. In this way, for every $S$, the equality $u_S = \sum_{r \in S} u_{r,S}$ holds and for every $r$, the equality $u_r = \sum_{S \in \wp(K), r \in S} u_{r,S}$ holds too. Thus the formula (26) becomes true under our extended $\mathcal{I}$. $\dashv$