Built-in Arithmetic in Knowledge Representation Languages

Shahab Tasharrofi and Eugenia Ternovska

Simon Fraser University, Canada
{sta44,ter}@cs.sfu.ca

Abstract. In previous papers, results about capturing the complexity class NP as model expansion task for languages with built-in arithmetic have been demonstrated. The purpose of this paper is to show implications of those results to the practice of KR, and, in particular, to two system languages of ASP (Gringo and Lparse) and the IDP system. In addition, we describe a logic which we call PBINT and show that PBINT provides a good basis for an attractive modelling language. We demonstrate that PBINT allows for compact and natural encodings.

1 Introduction

Currently, several different specification/modelling languages are being developed in different communities. They have their associated solvers, and are intended as universal languages for search problems in some complexity classes, usually NP (e.g. scheduling, planning, etc.). Examples include Answer Set Programming languages (Gringo, Lparse), modelling languages from the CP community such as ESSENCE [FGJ+05], or the input language of the IDP system[1] [WM09]. These languages do not closely correspond to first-order (FO) logic – they often contain inductive definitions and built-in arithmetic. Designers usually focus on the convenience of the language, and rarely pay attention to the expressiveness. For each language, several tasks can be studied – satisfiability and model checking are among them. Here, since we are interested in search problems, we focus on the task of *model expansion (MX)*, the logical task of expanding a given structure with new relations. The user axiomatizes their problem in some logic $\mathcal{L}$ (a specification/modelling language). The task of model expansion for $\mathcal{L}$ (abbreviated $\mathcal{L}$-MX), is:

Model Expansion for logic $\mathcal{L}$
Given: *(1)* An $\mathcal{L}$-formula ϕ with vocabulary $\sigma \cup \varepsilon$ and *(2)* A structure $\mathcal{A}$ for σ
Find: an expansion of $\mathcal{A}$, to $\sigma \cup \varepsilon$, that satisfies ϕ.

Thus, we expand the structure $\mathcal{A}$ with relations and functions to interpret ε, obtaining a model $\mathcal{B}$ of ϕ. The complexity of this task obviously lies in-between that of model checking (the entire structure is given) and satisfiability (no part of a structure is given). In the *combined setting* an instance consists of a structure together with a formula. We

[1] A language and system based on an extension of first-order logic with inductive definitions under well-founded semantics.

focus here on the *data complexity*, where the formula is fixed and the input consists of an instance structure only. We call σ, the vocabulary of $\mathcal{A}$, the *instance* vocabulary, and $\varepsilon := vocab(\phi) \setminus \sigma$ the *expansion* vocabulary.

Remark 1. Since FO MX can specify exactly the problems that ∃SO can, one might ask why we don't stick with the standard notion of ∃SO model checking. Primarily it is because we rarely use pure FO, and because for each language $\mathcal{L}$, MX is just one among several tasks of interest.

The authors of [MT05] emphasized the importance of *capturing NP* and other complexity classes for such languages. The capturing property is of fundamental importance as it shows that, for a given language:

(a) *we can express all of NP* – which gives the user an assurance of universality of the language for the given complexity class,

(b) *no more than NP can be expressed* – thus solving can be achieved by means of constructing a universal poly-time reduction (called *grounding*) to an NP-complete problem such as SAT or CSP.

The authors proposed to take the capturing property as a *fundamental guarding principle* in the development and study of declarative programming for search problems in this complexity class, and started careful development of foundations of modelling languages of search problems based on extensions and fragments of FO logic. While the current focus is on the complexity class NP, by no means, do we suggest that the expressive power of the languages for search problems should be *limited* to NP. Our goal is to design languages for non-specialist users who may have no knowledge of complexity classes. The users will be given a simple syntax within which they are *safe* and would be encouraged to express their problems in that syntax.

The classic Fagin's theorem [Fag74], relating ∃SO and NP, states that when the formula is fixed, FO MX captures NP. However, FO is too limiting for practical specification languages. Fagin's theorem allows one to represent all problems in NP, however, there is no direct way to deal with numbers and operations on them since in logic we have abstract domain elements – encoding of numbers may be needed. A usable logic for these problems would use standard arithmetic, as in all realistic modelling languages.

One of the main contributions of this paper is to formally investigate the expressive power of arithmetic in some of the practical modelling languages such as the ASP language of Gringo and Lparse systems and the system language of IDP system. For example, as Section 3 shows, if your system uses only ordinary arithmetical operations such as $+$, $\times$ and ordinary arithmetical aggregates such as Σ, Π, min and max, then you will not be able to axiomatize some common computational problems (such as integer factorization) naturally. i.e., you would need to use e.g. binary encodings of numbers. In these cases, you might need to use binary encodings as opposite to using built-in arithmetic.

These limitations apply regardless of whether fixpoint constructions are present in the language or not. For example, both ASP language of Gringo and Lparse and the system language of IDP system support fixpoints while still unable of naturally axiomatizing integer factorization. Section 3 proves this by introducing several categories

of logics and showing that all the system languages above fall into these categories. It is proved that, under complexity assumptions, these categories cannot express some problems naturally.

Another motivation in this paper is to design a logic where you could use arithmetic naturally, and where you can even have some form of quantification over the infinite domain of integers. We want to impose as few limitations as possible, but some are of course needed. In particular, we need to limit the range of the quantifiers. We also need to limit the number of elements appearing in the solutions. This is done by predicates which we call guards. By itself, this restriction is obvious and does not appear to be a contribution, especially given that most practical languages do exactly that. What is new here is that *the exact choice of the arithmetic operations and the kind of guards allowed matter a lot*.

A solution was given in [TT10]. There, we introduce a new set of arithmetical operations and show that, by supporting these operations, you can capture exactly NP over arithmetical structures, i.e., (1) nothing outside NP is expressible and (2) for every problem in NP involving arithmetic (and numbers), there is specification Φ that works with numbers as numbers (instead of encoding them using abstract domain elements). This paper introduces this logical fragment in Section 4 but does not give proofs of capturing NP.

Later on, in Section 5, we give some examples (including those that could not be axiomatized in other languages) and show that the new logic can axiomatize them naturally using its built-in arithmetical operators. The fragment can be viewed as an idealized specification language, and, hopefully, inspire the development of practical KR languages.

2 Background: MX with Arithmetic

Throughout the paper, we use $\exists\bar{x}$ to denote $\exists x_1 \ldots \exists x_n$ and $\forall\bar{x}$ to denote $\forall x_1 \ldots \forall x_n$.

Embedded MX Embedded finite model theory (see [Lib04,Lib07]), the study of finite structures with domain drawn from some infinite structure, was introduced to study databases containing numbers and numerical constraints. Rather than a database being a finite structure, we take it to be a set of finite relations over an infinite domain.

Definition 1. *A structure $\mathcal{A}$ is* embedded *in an infinite* background *(or* secondary*) structure $\mathcal{M} = (U; \bar{M})$ if it is a structure $\mathcal{A} = (U; \bar{R})$ with a finite set $\bar{R}$ of finite relations and functions, where $\bar{M} \cap \bar{R} = \emptyset$. The set of elements of U that occur in some relation of $\mathcal{A}$ is the* active domain of $\mathcal{A}$*, denoted* $adom_{\mathcal{A}}$.

In database research, embedded structures are used with logics for expressing queries. Here, we use them similarly, with logics for MX specifications. Throughout, we use the following conventions: σ denotes the vocabulary of the embedded structure $\mathcal{A} = (U; \bar{R})$, which is the instance structure; ν denotes the vocabulary of an infinite background structure $\mathcal{M} = (U; \bar{M})$; ε is an expansion vocabulary; A formula ϕ over $\sigma \cup \nu \cup \varepsilon$ constitutes an MX specification. The model expansion task remains the same: expand a (now embedded) σ-structure to satisfy ϕ.

GGF$_k$ Logical Fragment The authors of [TM09] use a guarded logic in an embedded setting, which allows them to quantify over elements of the background structure (unlike, e.g. [GG98]). To do so, they use an adaptation of the guarded fragment GF$_k$ of FO [GLS01]. In formulas of GF$_k$, a conjunction of up to k atoms acts as a *guard* for each quantified variable.

Definition 2. *The* k-guarded fragment *GF$_k$ of FO (with respect to σ) is the smallest set of formulas that:*

1. *contains all atomic formulas;*
2. *is closed under Boolean operations;*
3. *contains $\exists\bar{x}\,(G_1 \wedge \ldots \wedge G_m \wedge \phi)$, provided the G_i are atomic formulas of σ, $m \leq k$, $\phi \in GF_k$, and each free variable of ϕ appears in some G_i.*
4. *contains $\forall\bar{x}\,(G_1 \wedge \ldots \wedge G_m \supset \phi)$ provided the G_i are atomic formulas of σ, $m \leq k$, $\phi \in GF_k$, and each free variable of ϕ appears in some G_i.*
   For a formula $\psi := \exists\bar{x}\,(G_1 \wedge \ldots \wedge G_m \wedge \phi)$, conjunction $G_1 \wedge \ldots \wedge G_m$ is called the existential guard *of the tuple of quantifiers $\exists\bar{x}$;* universal guard *is defined similarly.*

Example 1. Let ε be $\{E_1, E_2\}$. The following formula is not guarded: $\forall x \forall y\,(E_1(x,y) \supset E_2(x,y))$. It is guarded when E_1 is replaced by P which is not in ε. The following formula is the standard encoding of the temporal formula $Until(P_1, P_2)$: $\exists v_2\,(R(v_1,v_2) \wedge P_2(v_2) \wedge \forall v_3\,(R(v_1,v_3) \wedge R(v_3,v_2) \supset P_1(v_3)))$. The formula is 2-guarded, i.e., is in GF$_2$, but it is not 1-guarded.

The guards of GF_k are used to restrict the range of quantifiers. They also use "upper guard" axioms, which restrict the elements in expansion relations to those occurring in the interpretation of guard atoms. To formalize this, they introduce the following restriction of FO, denoted GGF$_k(\varepsilon)$.

Definition 3. *The double-guarded fragment* GGF$_k(\varepsilon)$ *of FO, for a given vocabulary ε, is the set of formulas of the form $\phi \wedge \psi$, with $\varepsilon \subset vocab(\phi \wedge \psi)$, where ϕ is a formula of GF_k, and ψ is a conjunction of* upper guard axioms*, one for each symbol of ε occurring in ψ, of the form $\forall\bar{x}\,(E(\bar{x}) \supset G_1(\bar{x}_1) \wedge \cdots \wedge G_m(\bar{x}_m))$, where $m \leq k$, and the union of free variables in the G_i is precisely $\bar{x}$.*

Guards of GF$_k$, that restrict quantifier ranges, are *lower guards*, and guards of Def. 3 are *upper guards*. In GGF$_k$, all upper and lower guards are from the instance vocabulary σ, so ranges of quantifiers and expansion predicates are explicitly limited to $adom_{\mathcal{A}}$.

To finish definition of the logic, they define well-formed terms which depends on the vocabulary of the background structure. The authors of [TM09] use *arithmetical structures*, same as [GG98]. They also introduce a fragment of arithmetical structures known as *small cost arithmetical structures*.

They prove that GGF$_k$ captures NP for small cost arithmetical structures[2]. This paper continues on their path and proves that the property of capturing NP over small cost arithmetical structures can be extended to several practical KR languages.

[2] We use the definition of arithmetical structures and small cost arithmetical structures in this paper. So, for presentation reasons, these definitions are moved from background section to where they are used in Section 3.1.

3 Capturing and Non-Expressibility Results for Practical KR Languages

This section studies the expressiveness of built-in arithmetic in existing KR languages. But, in order to formally investigate the expressibility of KR languages, we first need to define what we mean by a specific language. We have based our investigations on system manuals published for these languages. And, as these manuals often neglect the details, we have had to do several experiments in order to understand if particular specifications are allowed or not. In the end, we believe that our results are true of the specific languages that we will talk about.

As is the case with all practical languages, they evolve over time and their syntax changes from one version to another. So, what is not allowed in current version, may be allowed in the next version or vice versa. Therefore, we specify the exact manual and version of these languages as below:

1. IDP version 1.4.4 and the accompanying manual published in August 2009 [WM09].
2. Gringo version 2.0.5 and the accompanying manual published in November 2008 [GKK$^+$08].
3. Lparse version 1.1.2 and the accompanying Lparse 1.0 user's manual [Syr00].

3.1 Capturing Results

Definition 4. *An* Arithmetical structure *is a structure $\mathcal{N}$ containing at least*

$$(\mathbb{N}; 0, 1, \chi, <, +, ., min, max, \Sigma, \Pi)$$

with domain $\mathbb{N}$, the natural numbers, and where min, max, Σ and Π are multi-set operations[3] *and $\chi[\phi](\bar{x})$ is the characteristic function. Other functions, predicates, and multi-set operations may be included, provided every function and relation of $\mathcal{N}$ is polytime computable.*

Definition 5. *For an embedded arithmetical structure $\mathcal{D}$, define $cost(\mathcal{D})$, the cost of $\mathcal{D}$, to be $\lceil \log_2(l+1) \rceil$, where l is the largest number in $adom_{\mathcal{A}}$.*

Definition 6 (Small cost structures). *A class $\mathcal{K}$ of embedded arithmetical structures has small cost if there is some $k \in \mathbb{N}$ such that $cost(\mathcal{D}) \leq |adom_{\mathcal{D}}|^k$, for every $\mathcal{D} \in \mathcal{K}$.*

Next, we are going to give a theorem that relates the expressibility power of ASP over arithmetical structures to the expressive power of GGF$_k$ fragment of logic. However, ASP programs are traditionally defined over relational structures and, thus, we have to extend this notion to arbitrary structures. In order to do this, we define *λ-restricted ASP Programs*.

Definition 7 (λ-restricted ASP Programs). *Let $B(v, r)$ denote the set of predicate symbols that appear in the body of rule r with variable v as a term in it, i.e., the term consists of only variable v. Also, let $V(r)$ denote all the free variables in r and $R_{\Pi}(p)$*

[3] Multi-sets are generalizations of sets that allow multiple occurrence of elements.

denote all rules in ASP program Π with predicate symbol p in their head. Also, let $M(r)$ denote all multi-set terms t of r and $V_m(m)$ denote all variables that are quantified by multiset operation m and $B_m(v, m)$ denote all predicate symbols that appear as a positive atom in multi-set operation m and with variable v as one of their arguments. We say that an ASP program Π is λ-restricted *if there is a function λ from predicate symbols of Π to natural numbers such that for all predicate symbols $\mathcal{S}$ in vocabulary of Π:*

$$\max\{\min\{\lambda(\mathcal{T}) \mid \mathcal{T} \in B(v,r)\} \mid v \in V(r),\ r \in R_\Pi(\mathcal{S})\} < \lambda(\mathcal{S}), \text{ and,}$$
$$\max\{\min\{\lambda(\mathcal{T}) \mid \mathcal{T} \in B_m(v,m)\} \mid v \in V_m(m),\ m \in M(r),\ r \in R_\Pi(\mathcal{S})\} < \lambda(\mathcal{S})$$

Note that, although the motivation for Definition 7 comes from Gringo, it is in fact different from both λ-restrictedness in [GST07] and level-restrictedness in [GKK$^+$08]. To see why Definition 7 is different from λ-restrictedness defined in [GST07], look at the following example which is not accepted by Definition 7 but is λ-restricted due to [GST07]:

$$q(0).$$
$$p(x) \leftarrow q(0 \times x).$$

Also, to see why Definition 7 is different from level-restrictedness defined in [GKK$^+$08], observe that assignment operator is not present in Definition 7. In fact, Definition 7 accepts a subclass of ASP programs that are characterized by level-restrictedness property which is, in turn, generalized to safety property in newer versions of Gringo software. The reason we use this limited class of ASP programs, and not the more inclusive versions, is to define a subclass of ASP programs that remains inside NP in the presence of arithmetic constructs. This way practical ASP solvers can define a switch to either limit users to this syntax and give them a performance guarantee in return (because of remaining in NP) or to give them full access to the ASP language (which can describe problems outside NP) but without such performance guarantees.

Now, we can use the notion of λ-restricted ASP programs and define admissible ASP programs over arithmetical structures to be the set of λ-restricted ASP programs over such background structures.

Theorem 1. *Let $\mathcal{K}$ be a class of small-cost embedded arithmetical structures over vocabulary $\sigma \cup \epsilon \cup \nu$. Then the following are equivalent:*

1. *$\mathcal{K} \in$ NP;*
2. *There is a first order formula ϕ of GGF$_k(\varepsilon)$ such that $\mathcal{D} \in \mathcal{K}$ if and only if there is an expansion $\mathcal{D}'$ of $\mathcal{D}$ to ε so that $\mathcal{D}' \models \phi$;*
3. *There is a λ-restricted ASP program P with instance vocabulary σ such that $\mathcal{D} \in \mathcal{K}$ iff there is an expansion $\mathcal{D}'$ of $\mathcal{D}$ so that $\mathcal{D}'$ is a stable model for P.*

Proof. $(1) \Rightarrow (2)$ is shown in [TM09].

$(2) \Rightarrow (3)$ is shown by Lloyd-Topor transformation. We first push all negations inside and then introduce new relation symbols for negated expansion predicates. For example, for expansion predicate $E(\bar{x})$, new relation symbol $E'(\bar{x})$ is introduced and two following sentences are added to the ASP program:

$$E(\bar{x}) \leftarrow \text{ not } E'(\bar{x}).$$
$$E'(\bar{x}) \leftarrow \text{ not } E(\bar{x}).$$

Also, new relation symbols are introduced for each sub-formula and Lloyd-Topor transformation is used to relate these sub-formulas together in an appropriate way. However, the resulting ASP program is not still λ-restricted.

In order to convert this ASP program into a λ-restricted ASP program, we have to incorporate information from lower and upper guards in the GGF_k specification. Such guards define a permissible domain for each sub-formula. Therefore, we introduce new domain predicates based on this information. These predicates can be defined using a completely positive and non-recursive ASP program. So, all rules generated above can be modified so as their variables are bounded by these new domain predicates.

Note that, here, we do not claim that a first order formula ϕ can be translated into a λ-restricted ASP program via Lloyd-Topor transformation. What we claim (and what is needed here for the proof) is that, informally, for a specification ϕ in $\text{GGF}_k(\varepsilon)$, there is a λ-restricted ASP program P such that for all instance structures $\mathcal{D}$, there is a certificate for $\mathcal{D}$ in the $\text{GGF}_k(\varepsilon)$ sense iff there is a certificate for $\mathcal{D}$ in the ASP sense.

The key here is that the transformation does not have to preserve the entire models of the GGF_k formula. What is needed to be presereved is the existence of an expansion (and not the equivalence of the set of expansions) for a given instance structure.

Also, the expansion vocabulary of the ASP program does not have to be the same as ε.

$(3) \Rightarrow (1)$ is shown by giving a machine in NP that first guesses a stable model and then checks its stability in polytime. The existence of such a guessing procedure is guaranteed by the λ-restrictedness property of the ASP program.

Corollary 1. *ASP language of Lparse and Gringo captures small cost arithmetical NP problems.*

Corollary 2. *The IDP language captures the small cost arithmetical NP problems.*

Proof. By Theorem 1, GGF_k covers all small cost NP structures. However, we know that except for characteristic function χ, IDP supports all the rest of GGF_k. So, we only need to show that χ can be written in terms of other arithmetical functions. This is easy to show: $\chi[\phi] \equiv \Sigma\{1 : \phi\}$.

3.2 Non-expressibility Results

This part considers some natural arithmetical problems and shows that they cannot be encoded using only built-in arithmetic of ASP languages or the input language of the IDP system. Two such problems are considered: the Integer Factorization problem and the Quadratic Residue Problem. We first define them:

Integer Factorization is a natural arithmetical problem with wide applications in cryptography. You are given a number n and asked to either report it as a prime or find one of its nontrivial factors (a positive factor except 1 and n itself). As a side-note, although checking primality is in polytime, it is not known if finding nontrivial factors of a composite number can also be done in polytime. Also, there are several other versions of factorization problem which are all polynomially equivalent, i.e., using an oracle for one version, solution to other versions can be found in polytime.

Quadratic Residue Problem is an NP-complete problem that involves only a few numbers. You are given three numbers n, a and c and you are asked if there is a number x such that $x \geq c$ and $x^2 \equiv a \pmod{n}$.

Various domain-restricted logical fragments Now, we define several logical fragments by restricting their MX tasks. We also show that some practical KR system languages such as ASP and IDP fall into these fragments. Then, by showing some non-expressibility results for these logical fragments, we effectively prove that system languages of ASP and IDP cannot express integer factorization and quadratic residue problems naturally.

Definition 8 (Active-domain-restricted logical fragment). *Let $\mathcal{L}$ be a logical fragment so that for all specifications ϕ in $\mathcal{L}$, and for all arithmetical structures $\mathcal{A}$ over σ (instance vocabulary) and all expansions of $\mathcal{A}$ such as $\mathcal{B}$ satisfying ϕ, we have that $adom_{\mathcal{B}} = adom_{\mathcal{A}}$. We call this logical fragment an* active-domain-restricted logical fragment.

Definition 9 (Polytime domain-restricted logical fragment). *Let $\mathcal{L}$ be a logical fragment so that for any specification ϕ in $\mathcal{L}$, there is a monotone polytime computable mapping $P : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ with the following property: For all arithmetical structures $\mathcal{A}$ over σ (instance vocabulary) and for all structures $\mathcal{B}$ expanding $\mathcal{A}$ and satisfying ϕ, we have $adom_{\mathcal{B}} \subseteq P(adom_{\mathcal{A}})$. We call such logical fragments a* polytime domain-restricted logical fragment.

Definition 10 (Loosely domain-restricted logical fragment). *Let $\mathcal{L}$ be a logical fragment so that for any specification ϕ in $\mathcal{L}$, there is a function $f : \mathbb{N} \to \mathbb{N}$ and a monotone polytime mapping $P : 2^{\mathbb{N}} \times 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ such that: For all arithmetical structures $\mathcal{A}$ over σ (instance vocabulary) and for all structures $\mathcal{B}$ expanding $\mathcal{A}$ and satisfying ϕ, we have $adom_{\mathcal{B}} \subseteq P(adom_{\mathcal{A}}, \{1, 2, 3, \cdots, f(|adom_{\mathcal{A}}|)\})$. We call such logical fragments a* loosely domain-restricted logical fragment.

Remark 2. Every polytime domain-restricted logical fragment is also a loosely domain-restricted logical fragment with f being a constant function and P being the same as before except it takes two argument and neglects the second one.

However, there is an important difference between these two fragments. Namely, when specification is fixed, for polytime domain-restricted logical fragments, polytime grounding (in the size of instance structure) is ensured; while, for loosely domain-restricted logical fragments, such grounding cannot be guaranteed.

Proposition 1. *The language of the IDP system is a domain-restricted logical fragment.*

Proof. All predicates and functions in IDP system language should have proper type names. Also, all types are given as part of the input. Thus, the active domain of all structures satisfying an specification in IDP is exactly the active domain of the instance structure.

Proposition 2. *The language of ASP (without Σ) accepted by Gringo and Lparse is a polytime domain-restricted logical fragment.*

Proof. For Gringo, we know that a correctly constructed ASP program should be level-restricted. So, we use induction on the levels of an ASP program and show that, for each level l, there is a monotone polytime program $P_l : 2^{\mathbb{N}} \rightarrow 2^{\mathbb{N}}$ that, given the active domain of the previous level, generates the active domain of the new level. So, as any fixed ASP specification has only constantly many different levels, we can combine all P_l's to obtain a new monotone polytime program P that satisfies the polytime domain-restrictedness condition.

For Lparse, we just use the fact that all Lparse programs are also Gringo programs [GKK$^+$08].

The condition on not including Σ in Proposition 2 is essential because, adding it will enable us to describe predicates consisting of exponentially many values. The following ASP program shows one such scenario:

$$
\begin{array}{l}
V(2^i), \text{ for } i \in \{0, 1, 2, \cdots, n-1\}. \\
A(X) \leftarrow \text{not } A'(X), V(X). \\
A'(X) \leftarrow \text{not } A(X), V(X). \\
E(X) \leftarrow X = \Sigma(Y; A(Y), V(Y)).
\end{array}
$$

This program has domain size n, but predicate E has domain 0 to $2^n - 1$.

Proposition 3. *ASP language of Gringo and Lparse is a loosely domain-restricted logical fragment.*

Proof. For Gringo, again, we use the level-restrictedness property. Assume that your specification ϕ has l levels. As discussed above, each summation can (potentially) exponentiate the size of the domain. But, by level-restrictedness property, summations separate levels. Thus, at most l different exponentiations can occur. Now, if we take f as follows, it gives an upper bound on the number of elements at level l:

$$
f(n) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}}_{l \ times}
$$

Then, we use the same monotone polytime program P as in Proposition 2 with the difference that it takes two arguments and neglects the second one. This program runs in polytime because the size of its input is so large that all of its computation time is bounded by a polynomial in that (large) number. Please note that the function f given above is a very rough upper bound and we believe that upper bounds of form $f(n) = 2^{poly(n)}$ work too (although with a more detailed proof).

For Lparse, again, we only use the fact that all Lparse programs are also Gringo programs [GKK$^+$08].

Non-expressibility for domain-restricted logical fragments We now prove that integer factorization and quadratic residue problem can not be naturally axiomatized in the logical fragments we defined. For active-domain-restricted logical fragments, this is obvious. We need new numbers except those in the domain. Below, we prove the same property for the two other domain-restricted fragments, i.e., polytime and loosely domain-restricted logical fragments.

Theorem 2. *If $\mathcal{L}$ is a polytime or loosely domain-restricted logical fragment, then*

1. *$\mathcal{L}$ cannot express integer factorization using built-in arithmetic unless factorization is in polytime.*
2. *$\mathcal{L}$ cannot express quadratic residue problem using its built-in arithmetic unless P=NP.*

Proof. **(for polytime domain-restricted logical fragments)** Let $\mathcal{L}$ be a polytime domain-restricted logical fragment and ϕ be a specification in such a fragment. Then, by definition, there is a monotone polytime program P that gives an upper bound on the active domain of all valid expansions of a given instance structure.

Now, if ϕ axiomatizes integer factorization with its built-in arithmetic, at least one factor of number n should appear in the output of P. So, checking all numbers that P outputs gives us one such factor. Also, as P is polytime, the whole checking procedure would also be polytime and we would solve integer factorization in polytime.

Similarly, if ϕ axiomatizes quadratic residue problem using its built-in arithmetic, then x appears in the output produced by P. So, checking all outputs of P gives us such x if it exists. Again, P being polytime implies that the whole procedure is also polytime and P=NP (because quadratic residue problem is NP-complete).

Proof. **(for loosely domain-restricted logical fragments)** Let $\mathcal{L}$ be a loosely domain-restricted logical fragment and ϕ be a specification in such a fragment. Then, by definition, there is a function f and monotone polytime program P such that the active domain of all valid expansions $\mathcal{B}$ of an input structure $\mathcal{A}$ are upper-bounded by $P(adom_{\mathcal{A}}, \{1, 2, 3, \cdots, f(|adom_{\mathcal{A}}|)\})$.

However, in the case of the two problems above (factorization and quadratic residues), we know that the number of elements in the input structure is always constant (one element in factorization and 3 elements in quadratic residue). Therefore, $f(|adom_{\mathcal{A}}|)$ is always a constant (depending on f but not $\mathcal{A}$). Thus, the set $\{1, 2, 3, \cdots, f(|adom_{\mathcal{A}}|)\}$ does not depend on structure $\mathcal{A}$. So, program P takes the active domain of $\mathcal{A}$ and returns a set S which upper-bounds the active domain of all valid expansions of $\mathcal{A}$. The rest of the proof can be carried out in the same way as the previous case.

Corollary 3. *Using only built-in arithmetic of ASP language of Gringo and Lparse (with or without Σ),*
(1) Unless P=NP, the quadratic residue problem cannot be axiomatized naturally, and
(2) Axiomatizing integer factorization naturally is possible only if it is polytime computable.

So, we proved that the two problems of factorization and quadratic residue cannot be axiomatized in either ASP or IDP system languages. However, we disregarded one

of the features of both languages we analyzed: the compact domain representation, i.e., the use of two numbers n_1 and n_2 to compactly represent all numbers in the range between n_1 and n_2. In fact, the compact domain representation can be used to naturally axiomatize both problems above. However, there are two drawbacks associated with using compact domain representations:

1. First, the compact domain representations would take one well beyond NP and would enable one to describe, for example, the NEXP-complete problem of tiling. Therefore, any hope for polynomial time grounding would be lost.
2. Second, the compact domain representation is not really an option for any reasonable size instance. For example, in the case of factorization, the numbers one need to factor in practice need more than 200 digits to represent. So, using compact domain representation yields a domain of size 10^{200} which cannot be stored in any reasonable size memory (in fact, it is more than the number of atoms in the observable universe).

4 Logic PBINT

In this section, we describe a logic that can unconditionally characterize NP problems involving arithmetic. The first step towards this goal is to use a different background structure:

Definition 11. *A* Compact Arithmetical structure *is a structure $\mathcal{N}^c$ having at least $(\mathbb{N}; 0, 1, +, \times, <, ||\ ||)$ with domain $\mathbb{N}$, the natural numbers, where 0, 1, $+$, $\times$ and $<$ have their usual meaning and $||x||$ returns the size of binary encoding of number x, i.e., $||x|| = 1 + \lfloor log_2(x+1) \rfloor$. Other functions, predicates, and multi-set operations (min, max etc.) may be included, provided every function and relation of $\mathcal{N}^c$ is polytime computable.*

Requirements on σ As before, we consider embedded MX, but the embedding is into the compact arithmetical structure. We make some assumptions about the instance vocabulary σ. It contains predicate $adom_{\mathcal{A}}$ and a constant $SIZE$. The constant $SIZE$ is equal to $|adom_{\mathcal{A}}| \times S$ where $|adom_{\mathcal{A}}|$ is the number of elements in the active domain and S is the size of binary encoding of the maximum element of the active domain. In other words, $SIZE$ upper-bounds the number of bits needed to encode (in binary) the input structure $\mathcal{A}$ embedded in $\mathcal{N}^c$. We also need a constant *default* denoting a particular default value needed in upper guards on functions. Its meaning is specified by the user.

Logic PBINT We introduce a new logic, PBINT, standing for Polynomially Bounded Integers. This logic is a variant of the double-guarded logic except we use compact arithmetical structures and allow function symbols in both σ and ε, or new kinds of guards, with more freedom in existential and upper guards on the outputs of expansion functions. The three forms of guards in PBINT are as follows:

1. **Instance Guards** are instance relations (including $adom_{\mathcal{A}}$) interpreted by the instance structure $\mathcal{A}$. Note that, although not required to be so, all specifications can be rewritten so as they only use $adom_{\mathcal{A}}$ as an instance guard.
2. **Polynomial Range Guards** are relations of the form $poly_1(SIZE) \leq x \leq poly_2(SIZE)$ with $poly_1$ and $poly_2$ two polynomials.

3. **PBINT Guards** are relations of form $||x|| \leq poly(SIZE)$ where $poly(SIZE)$ is a polynomial depending only on the constant $SIZE$.

Instance guards and polynomial range guards define ranges of size at most polynomial in the binary encoding size of structure. However, PBINT guards can define ranges with exponentially many different integers. For example, condition $||x|| \leq SIZE$ is equivalent to $x \leq 2^{SIZE-1} - 1$, exponential in the value of $SIZE$. Also, note that guards definable by stratifiable inductive definitions with (1), (2) as the base cases can be added without changing our results.

Definition 12 (logic PBINT). *We define our logic as follows.*
Background Structure: *the compact arithmetical structure.*
Terms *are constructed as usual over* $\nu \cup \sigma \cup \varepsilon$.
Formulas:

(a) **Upper Guards**
  - *i. Expansion* relations *are upper-guarded by instance or polynomial range guards.*
  - *ii. An expansion* function f *has an upper guard of form* $\forall \bar{x} \forall y\ (f(\bar{x}) = y \Rightarrow (G(\bar{x}, y) \vee y = default))$ *where* $G(\bar{x}, y)$ *is a conjunction of guards jointly guarding variables* $\bar{x}$ *and* y *so that* $\bar{x}$ *is upper-guarded by instance or polynomial range guards and* y *is upper-guarded by any of the three types of guards.*

(b) **Lower Guards**
  - *i. Existential guards: any of the three types of guards.*
  - *ii. Universal guards: instance or polynomial range guards.*

The constant "default" can be interpreted by any number at the user's choice. The part $y = default$ in *(a(ii))* above is needed because all functions in FO logic are total, thus defined on all natural numbers. Without that part, the upper guard axioms on expansion functions would always be false making all specifications with such functions useless.

Functions have meaningful (non-default) outputs on a finite number of inputs. Thus, we can obtain a finite representation (encoding) of instance and expansion structures.

Having functions in the instance vocabulary imposes a small inconvenience with the definition of the active domain. In a formalism based on classical logic, all terms are total, and therefore defined on all integers. Thus, if we add functions, the notion of an active domain becomes meaningless. On the other hand, the user might (quite reasonably) assume that the inputs and the outputs of the instance functions are from a finite domain, which makes these functions to be non-total. A safe solution would be to advise the user to use graphs of functions (i.e., the corresponding predicates) instead of instance functions. It would solve the problem with the definition of an active domain, but this solution seems too limiting for a nice logic. Instead, we choose to allow instance functions, but to require all instance functions to have upper guards and the "default" value for inputs outside of the intended range, just as we have for expansion functions. Active domain now contains all elements of the universe contained in all instance relations, together with all elements in the ranges of the instance functions.

Capturing NP with PBINT-MX

Theorem 3. *Let* $\mathcal{K}$ *be an isomorphism-closed class of compact arithmetical embedded structures over vocabulary* σ. *Then the following are equivalent:*

1. $\mathcal{K} \in NP$,

2. *there is a PBINT sentence ϕ of a vocabulary $\tau = \sigma \cup \nu \cup \varepsilon$, such that $\mathcal{A} \in \mathcal{K}$ iff there exists an expansion $\mathcal{B}$ of $\mathcal{A}$ with $\mathcal{B} \models \phi$.*

We do not give any proof for Theorem 3 in this paper. The proof can be found in [TT10]. The importance of this theorem is in its ability to capture all NP with arithmetic. As shown previously, most of previous frameworks for arithmetic in KR suffer from the fact that they can only axiomatize certain arithmetical problems in NP. For example, we showed that ASP and IDP cannot axiomatize integer factorization using their built-in arithmetic. On the other hand, Theorem 3 shows that PBINT can axiomatize exactly those problems involving arithmetic which are in NP.

Moreover, the background structure of PBINT is much simpler than many background structures in practical KR languages. In particular, there is no built-in aggregate in compact arithmetical structures. Together with Theorem 3, it shows that PBINT can define aggregates in terms of more primary operations (those given in compact arithmetical structures). Therefore, adding aggregates to your language would not increase the expressibility of your language.

Thus, PBINT gives you a very concrete basis to build your language upon. It tells you that if your language somehow supports all PBINT constructs, you can (1) be sure that your language captures all of NP and (2) unless your other constructs are very powerful (outside NP $\cap$ co-NP), you can safely add them to your language without worrying about its complexity implications.

5 Examples of PBINT Axiomatizations

In this section, we give some examples of PBINT axiomatizations to demonstrate the naturality of them. These examples include the two examples of integer factorization and quadratic residue so as to contrast against the results of Section 3.

Example 2 (Disjoint Scheduling). Given a set of Tasks, $t_1, \cdots, t_n$ and a set of constraints, find a scheduling that satisfies all the constraints. Each task t_i has an earliest starting time $EST(t_i)$, a latest ending time $LET(t_i)$ and a length $L(t_i)$. There are also two predicates $P(t_i, t_j)$ and $D(t_i, t_j)$ which say, respectively, that task t_i should end before task t_j starts, and two tasks t_i and t_j cannot overlap. We are asked to find two functions $start(t_i)$ and $end(t_i)$ satisfying the given conditions.

In PBINT, we axiomatize this problem as follows: Instance vocabulary σ consists of symbols EST, LET, L, $Task$, P and D. Expansion vocabulary consists of two functions $start$ and end. The axiomatization below first gives the upper guards on these functions and then the axioms:

$$\forall t \forall s\ (start(t) = s \Rightarrow (Task(t) \wedge ||s|| \leq SIZE) \vee s = default),$$
$$\forall t \forall e\ (end(t) = e \Rightarrow (Task(t) \wedge ||e|| \leq SIZE) \vee e = default),$$
$$\forall t_i\ (Task(t_i) \Rightarrow start(t_i) \geq EST(t_i)),$$
$$\forall t_i\ (Task(t_i) \Rightarrow end(t_i) \leq LET(t_i)),$$
$$\forall t_i\ (Task(t_i) \Rightarrow start(t_i) + L(t_i) = end(t_i)),$$
$$\forall t_i \forall t_j\ (P(t_i, t_j) \Rightarrow end(t_i) \leq start(t_j)),$$
$$\forall t_i \forall t_j\ (D(t_i, t_j) \Rightarrow end(t_i) \leq start(t_j) \vee end(t_j) \leq start(t_i)).$$

In a practical language, upper and lower guards are defined by types and need not be given explicitly. Here, the predicate $Task$ is a type and functions $start$ and end are functions from type $Task$ to integer type. So, predicate $Task$ can disappear from the above sentences.

Example 3 (Integer Factorization). You are given a number n and asked to find some nontrivial factor of n. Here, σ only has constant n and ϵ only constant p which is upper-guarded as follows: $||p|| \leq SIZE$ (which abbreviates $\forall m\ (c = m \Rightarrow ||m|| \leq SIZE)$ in case of zero-ary (constant) expansion functions). Now, the axiomatization is: $p > 1 \wedge p < n \wedge \exists q\ (||q|| \leq SIZE \wedge p \times q = n)$.

Example 4 (Quadratic Residues). You are given numbers r, n and c and asked to find a number x such that $x^2 \equiv r\ (\mathbf{mod}\ n)$ and $x < c$. Here, instance vocabulary consists of constants n, r and c and expansion vocabulary only has constant x upper-guarded by sentence $||x|| \leq SIZE$. The axiomatization consists of two sentences $0 \leq x \wedge x \leq c \wedge x < n$ and $\exists q\ (||q|| \leq SIZE \wedge x \times x = q \times n + r)$.

6 Related Work

Research in databases over infinite structures can be traced back to the seminal paper by Chandra and Harel [CH80]. There are several follow-up papers with developments in several directions including [Top91,Suc98,GG98], and more recent [Grä07]. Topor [Top91] studies the relative expressive power of several query languages in the presence of arithmetical operations. He also investigates domain independence and genericity in such frameworks.

Another line of database-motivated work over infinite background structures is embedded model theory (See [Lib04,Lib07]). Work in this area generally reduces questions on embedded finite models to questions on normal finite models. An important result in this area is the natural-domain-active-domain collapse for ∃SO for embedded finite models, as well as other deep expressiveness results. The work also describes a notion of safety (through e.g. range-restriction) to achieve safety with many background structures, and connections between safety and decidability. The active domain quantifiers are similar to our proposal of lower guards, however our goal was to reflect what is used in practical languages, namely the so-called domain predicates of Answer Set Programming and type information from other languages. We have done it through the use of upper and lower guards. In general, research in database theory is mostly focused around computability and the expressive power of query languages, while our interest, following [Grä07] is in capturing complexity classes, but in connection with specification/modelling languages. We plan, however, to investigate the applicability of domain-independence, range-restrictedness and other notions from embedded model theory to practical modelling languages.

Grädel and Gurevich [GG98] studied logics over infinite background structures in a more general computer science context. They characterized NP for arithmetical structures under some small weight property, generalized to the small cost condition in [TM09] (see [TM09] for a more detailed discussion). While this condition corresponds to existing languages (as shown in Section 3.1), our work here gives an unconditional result for capturing NP in the presence of arithmetical structures, and thus is

a step forward in the development of such languages. Instead of controlling access to the background structure through the use of weight terms [GG98] , we rely on guarded fragments, which is much closer to practical specification languages.

The work we mentioned so far is the closest to our proposal, and was the most inspirational. The research on descriptive complexity in the embedded setting also includes the work of Grädel and Meer [GM96], as well as Grädel and Kreutzer [GK99]. Another line (Cook, Kolokolova and others [CK01]) establishes connections between bounded arithmetic and finite model theory, in particular by relying on Grädel's characterization of PTIME.

Another direction on capturing complexity classes is bounded arithmetic, including [Bus85,Ske05,BC92]. However, the characterization of complexity classes there is in terms of *provability* in systems with a limited collection of non-logical symbols, and is not applicable here.

Built-in arithmetic is implemented in many modelling languages, e.g. the IDP system [WM09], the Gringo system [GKK$^+$08], and LPARSE [Syr00]. There is also existing works such as [GOS09] and [BBG05] on how these systems deal with solving issues in presence of arithmetic constraints. However, we are not aware of any in-depth study of the expressiveness of such languages in the presence of arithmetic constraints. In many cases, allowing arithmetic constraints without careful restrictions provides the language with very high expressiveness, as is shown for ESSENCE [MT08].

7 Conclusion

In modelling languages, you are frequently faced with the problem of having a framework to support both natural specifications of problems, and reasoning about those problems. In this paper, we took our measure of naturality to be being able to use "built-in" arithmetic, and our measure of reasoning to be being in NP. We showed some examples of problems of practical importance and proved that several existing modelling languages cannot express these problems naturally (using their built-in arithmetic and not by encoding numbers using abstract domain elements). We also presented a solution to this problem and claimed (without giving the proof) that our fragment of logic can represent all arithmetical problems in NP naturally. We supported our claim by giving natural axiomatizations for problems in NP that could not be naturally axiomatized in existing modelling languages. This result guarantees universality of our logic for this complexity class and also settles our reasoning abilities by showing that all PBINT axiomatizations can be efficiently (in polytime) grounded to any state of the art solver of NP problems. Our work is a significant step forward from the previous proposal since it overcomes a number of limitations.

The language we proposed is natural because it is essentially FO logic, where guards can be made "invisible" through "hiding" them in a type system. Solving can be achieved through grounding to SAT, a work which is being performed in our group, but falls outside of the topic of this paper and this conference.

In summary, our work has pointed out some of the limitations of existing modelling languages and provided a solution to these limitations. Future directions include (a) analysis of other existing languages in connection with the logical fragments defined

here; (b) design of logics with different useful background structures and analysis of existing modelling languages with respect to these background structures, (c) continue with our implementation development.

Acknowledgement. This work is generously funded by NSERC, MITACS and D-Wave. We also express our gratitude towards the anonymous referees for their useful comments.

References

[BBG05] S. Baselice, P.A. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05)*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 2005.

[BC92] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 283–293, 1992.

[Bus85] S. R. Buss. *Bounded arithmetic*. PhD thesis, Princeton University, 1985.

[CH80] A. Chandra and D. Harel. Computable queries for relational databases. *Journal of Computer and System Sciences*, 21:156–178, 1980.

[CK01] S. Cook and A. Kolokolova. A second-order system for polytime reasoning based on grädel's theorem. In *Proceedings of Sixteenth Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 177–186, 2001.

[Fag74] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *Complexity of computation, SIAM-AMC proceedings*, 7:43–73, 1974.

[FGJ+05] A. M. Frisch, M. Grum, C. Jefferson, B. M. Hernandez, and I. Miguel. The essence of essence: A constraint language for specifying combinatorial problems. In *Proc. of the Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 73–88, 2005.

[GG98] E. Grädel and Y. Gurevich. Metafinite model theory. *Inf. Comput.*, 140(1):26–81, 1998.

[GK99] E. Grädel and S. Kreutzer. Descriptive complexity theory for constraint databases. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic, CSL '99, Madrid*, volume 1683 of *LNCS*, pages 67–81. Springer, 1999.

[GKK+08] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. *A User's Guide to gringo, clasp, clingo, and iclingo*, November 2008. http://potassco.sourceforge.net/.

[GLS01] G. Gottlob, N. Leone, and F. Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. In *PODS '01*, 2001.

[GM96] E. Grädel and K. Meer. Descriptive complexity theory over the real numbers. *Mathematics of Numerical Analysis: Real Number Algorithms*, 32:381–403, 1996.

[GOS09] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In P. Hill and D. Warren, editors, *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*, pages 235–249. Springer-Verlag, 2009.

[Grä07] E. Grädel. *Finite Model Theory and Descriptive Complexity*, pages 125–230. Springer, 2007.

[GST07] M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the Ninth*

International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 266–271. Springer-Verlag, 2007.

[Lib04] L. Libkin. *Elements of Finite Model Theory*. 2004.

[Lib07] L. Libkin. *Embedded Finite Models and Constraint Databases*, pages 257–338. Springer, 2007.

[MT05] D. G. Mitchell and E. Ternovska. A framework for representing and solving NP search problems. In *Proc. AAAI'05*, 2005.

[MT08] D. G. Mitchell and E. Ternovska. Expressiveness and abstraction in ESSENCE. *Constraints*, 13(2):343–384, 2008.

[Ske05] A. Skelley. *Theories and Proof Systems for PSPACE and the EXP-Time Hierarchy*. PhD thesis, University of Toronto, 2005.

[Suc98] D. Suciu. Domain-independent queries on databases with external functions. *Theor. Comput. Sci.*, 190(2):279–315, 1998.

[Syr00] T. Syrjänen. *Lparse 1.0 User's Manual*, 2000. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz.

[TM09] E. Ternovska and D. G. Mitchell. Declarative programming of search problems with built-in arithmetic. In *Proc. of 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 942–947, 2009.

[Top91] R. Topor. Safe database queries with arithmetic relations. In *Proc. 14th Australian Computer Science Conf*, pages 1–13, 1991.

[TT10] S. Tasharrofi and E. Ternovska. PBINT, a logic for modelling search problems involving arithmetic. In *Proceedings of the 17th Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'17)*, Yogyakarta, Indonesia, October 2010.

[WM09] J. Wittocx and M. Marien. *The IDP System*. KUL, August 2009. http://www.cs.kuleuven.be/˜dtai/krr/software/idpmanual.pdf.