On the Expressive Power of Essence

David G. Mitchell and Eugenia Ternovska

Computational Logic Laboratory
Simon Fraser University
{mitchell,ter}@cs.sfu.ca

Abstract. Essence is a syntactically rich language for specifying search and optimization problems. We show how to applying descriptive complexity theory to study the expressive power of Essence, and present several results on the expressive power of fragments incorporating certain features. The same approach can be applied to a number of other constraint specification or modelling languages. We believe this approach forms is an appropriate theoretical formalization for scientific development and study of constraint modelling languages and the tools and techniques used in their application.

1 Introduction

An important direction of work in constraint modelling is the development of declarative languages for specifying or modelling combinatorial search problems.[1] Examples of such languages (or tools with input languages that qualify) include: ASPPS [3] EaCL [12] ESRA [6] Essence [7] MidL [11] MXG [13, 14] NP-Spec [1] . An important role of these languages is to model *problems*, that is, to abstract away from individual instances of a problem. The user describes the general properties of their problem instances and solutions. Then, a solver takes as input a problem description (or "specification" or "model") together with an instance, and produces one or more solutions (if there are any). This is in contrast to much work in constraint-based solving where the user must either provide a reduction to a particular ground language (SAT, CSP) or must implement a solving strategy (eg., CLP).

These languages, if well-designed and supported by high quality tools, have the potential to greatly expand the range of applications and users of constraint solving technology. They also provide a convenient context in which to explore modelling options and reformulation techniques, as shown by the work of several groups including that of Frisch and his colleagues developing Essence and its companion tool Conjure, that of Mancini with the late Marco Cadoli on automatic reformulation of constraints, etc.

The general trend in the development of such declarative languages is toward greater abstraction and increased variety of syntactic features for user

[1] We do not distinguish specification and modelling here. The distinction is important, but is neither simple nor central to the results or point of this paper.

convenience. The design of these languages tend to be based on the designers experience and intuitions about what would be useful or convenient in practice, and the language features are formalized in different ways (or, in some cases, not at all). It is thus challenging to carry out a scientific study of the languages and their features. Such study would be useful in the design process, in comparison of languages, in the design of implementations supporting language use, and in identifying common ground that could be exploited by the broad community, for example in the form of standards, translation tools, etc. Our goal here is to take first steps in this direction, by beginning a study of one of the most fundamental questions about a formal language: What things can it say, or not? For languages which describe computational problems, the most basic form of such a question is: What is computational complexity of the problems the language can describe? We begin by looking at ESSENCE, which is probably the richest and most expressive language among those we have mentioned.

Our contributions here are the following:

1. Using ESSENCE as an example, we demonstrate that by adopting a logical formalization of problem specification, descriptive complexity theory can be used to formally study the expressive power of constraint modelling languages.
2. We examine the expressive power provided by various features of ESSENCE, which turns out to be very expressive. This provides material for thought for language designers, as expressiveness is a double-edged sword.
3. We point out that several other languages can be studied using the same approach. This, we believe, provides significant opportunities for the field in possibilities for, for example, standard base languages, automated translation or cross-compilation, theoretically well-founded studies of modelling and reformulation techniques and their application within broad classes of languages, etc.

Remark 1. Here, we distinguish expressiveness, in the formal sense of what problems can or cannot be specified with a language, from notions of expressiveness related only to user convenience. The latter is extremely important, but is not our subject here except to the extent that certain convenient features may lead to increased formal expressiveness.

In our study of ESSENCE, we begin with a small fragment of the language, and show that it can specify exactly the problems in NP. We then point out a number of extensions to this language that do not increase its expressive power. We point out one feature of ESSENCE, and many other languages, which gives them great expressive power – but which for most purposes we should ignore. Finally, we point out a number of features of the full language which give it great expressive power. Expressive power is, in and of itself, neither good nor bad. We believe it is useful to understand the expressive power of languages we design and use, and to be able to control and exploit this power. Obviously, we need a modelling language of sufficient expressive power to specify the problem at hand. On the other hand, expressiveness beyond what is needed, or certain

`given`	Vertex new type enum(. . .)
`given`	Colour new type enum(. . .)
`given`	Edge: rel Vertex × Vertex
`find`	Coloured: rel Vertex × Colour
`such that`	$\forall_{u:\mathrm{V}ertex}.\ \exists_{c:\mathrm{C}olour}.\ Coloured(u, c)$
`such that`	$\forall_{u:\mathrm{V}ertex}.\ \forall_{c1:\mathrm{C}olour}.\ \forall_{c2:\mathrm{C}olour}.\ (\neg(c1 = c2) \Rightarrow$ $\neg(Coloured(u, c1) \wedge Coloured(u, c2))$
`such that`	$\forall_{u:\mathrm{V}ertex}.\forall_{v:\mathrm{V}ertex}.\forall_{c:\mathrm{C}olour}.(Edge(u, v) \Rightarrow$ $\neg(Coloured(u, c) \wedge Coloured(v, c)))$

Fig. 1. An Essence specification of Graph Colouring

features that provide such expressiveness, may be an obstacle to effective design and implementation of solvers and other tools. We believe it is worthwhile to investigate the possibility of languages with restricted versions of these features which provide the desired convenience without the attendant expressive power.

2 A Logical Formalization

The tools we will apply – and propose to apply more generally in design and analysis of constraint modelling languages – are those of logic, and in particular finite model theory. Thus, we begin by adopting a logical point of view of the task at hand. All constrain modelling languages, while differing in styles and having their own particular goals, share with Essence the general goal of specifying search problems. We consider here what a specification of a search problem is from a logical point of view, in the particular context of Essence specifications.

2.1 A Little Essence

We will not attempt a full description of Essence. We will introduce features of Essence required to understand the paper as we need them, and these only by example or illustration. While this is sufficient to make the paper self-contained, for a fuller appreciation of Essence and our work we strongly encourage the reader to refer to [7].

Simple Essence specifications consist of three parts, identified by keywords `given`, `find`, and `such that`, which are suggestive of their roles. The "given" part specifies the types of objects which comprise instances; the "find" part specifies types of objects that comprise solutions; the "such that" part specifies the relationship between instances and their solutions.

Example 1. Figure 1 gives an Essence specification of Graph Colouring. The computational task described by this specification is this. We will be given a finite set $Vertex$, a finite set $Colour$, and a binary relation $Edge \subseteq Vertex \times Vertex$. Our task is to find a relation $Coloured \subseteq Vertex \times Colour$, which, in accordance with the constraints – the such that part – maps each $v \in Vertex$ to a unique colour so that no edge is monochromatic. $\diamond$

2.2 E_{FO}: A Small Fragment of Essence

Here we introduce a fragment of Essence that is convenient to explain our logical view and is a useful starting point for analysis.

Definition 1 (E_{FO}). Let E_{FO} denote the fragment of Essence defined by the following rules.

1. The "given" part consists of a sequence of statements of the following two forms:
   (a) `given` D new type enum(...), or
   (b) `given` R : rel $\mathrm{D}_1 \times \ldots \times \mathrm{D}_n$, where each D_i is a type declared by a previous given statement of form 1a, and $n \in \mathbb{N}$.
2. The "find" part consists of a sequence of statements each of the form
   `find` R : rel $\mathrm{D}_1 \times \ldots \mathrm{D}_n$
   where each D_i is the name of a type declared in a given statement of the form 1a, and $n \in \mathbb{N}$.
3. The "such that" part consists of a sequence of statements each of the form
   `such that` ϕ
   Here, ϕ is an expression which is syntactically a formula of function-free first-order logic, with two exceptions:
   (a) All quantifiers of E_{FO} are of the form $\exists_{x:\mathrm{D}}.$ or $\forall_{x:\mathrm{D}}.$, where x is a variable and D is a type declared by a given statement of form 1a.
   (b) We use $\Rightarrow$ for material implication in Essence expressions, and $\supset$ in formulas of classical logic.

The notation E_{FO} indicates the fragment of Essence corresponding to first order logic. E_{FO} is a kind of "kernel" of Essence of primarily theoretical interest – although many features used in typical specifications can be seen as abbreviations for more complex E_{FO} expressions.

2.3 Model Expansion as the Underlying Logical Task

The "given" part of an E_{FO} specification declares a vocabulary, or set of symbols, σ, with one symbol for each type or relation given by the instance. An instance gives an interpretation for each symbol of σ, which is a finite set of objects if the symbol was declared with form 1a of Definition 1, or a relation over some of the given types, if declared with form 1b. In logical terms, an instance is a finite structure for the vocabulary σ.[2] The universe of this finite structure is the union of all types given in the instance.

[2] A vocabulary is a set σ of relation and function symbols, each with an associated arity. A relational vocabulary has no function symbols. A structure $\mathcal{A}$ for vocabulary σ is a tuple containing a universe A, and a relation (function), defined over A, for each relation (function) symbol of σ. If R is a relation (a.k.a. predicate) symbol of

The "find" part of an E_{FO} specification also declares a vocabulary, ε. Interpretations of the symbols of ε are relations over the given types. If we take an instance σ-structure, and expand it by adding interpretations for ε, we have a finite structure of vocabulary $\sigma \cup \varepsilon$, consisting of an instance together with a "candidate solution". The role of the "such that" part of the specification is to identify those "candidate solutions" that are actual solutions.

Example 2. The specification of Graph Colouring in Figure 1 is an $\mathrm{E}_{FO}()$ specification. The instance vocabulary is $\sigma = (Vertex, Edge, Colour)$. An instance structure gives interpretations to the symbols of σ, in the form of the two sets $Vertex$ and $Colour$ and a binary relation $Edge \subseteq Vertex \times Vertex$. The domain of the structure will be the union of the sets of vertices and colours. The solution vocabulary is $\varepsilon = (Coloured)$, and a "candidate solution" is a binary relation over $Vertex \times Colour$. $\diamond$

2.4 FO Axioms for an E_{FO} Specification

In logical terms, the task specified by any E_{FO} specification is of the form: Given a finite σ-structure $\mathcal{A}$, find an expansion of $\mathcal{A}$ to $\sigma\cup\varepsilon$ which constitutes a solution. We now need to show that the remaining information contained in an E_{FO} specification can be accounted for in purely logical terms. To show we can do this, we first identify with each E_{FO} specification Γ a class $\mathcal{K}_\Gamma$ of finite $\sigma\cup\varepsilon$-structures containing exactly those structures consisting of an instance for Γ expanded with a (real, not candidate) solution. We then show that, corresponding to each E_{FO} specification Γ is a formula ϕ_Γ of first order logic whose finite models are exactly the structures in $\mathcal{K}_\Gamma$.

The information we need to express in our formula is the constraints, which comprise the "such that" part of the specification, and the type information. The constraints are just formulas of first-order logic, except for the type information in the quantifiers. It is easy to re-write these as pure FO formulas containing the same information. To do so, we recursively apply the following rules to each constraint.

– Rewrite $\forall_{x:D}.\ \phi(x)$ as $\forall x\ (D(x) \supset \phi(x))$,
– Rewrite $\exists_{x:D}.\ \phi(x)$ as $\exists x\ (D(x) \wedge \phi(x))$

For an E_{FO} specification Γ, denote by Γ_C the translation of the "such that" constraints to first-order formulas.

It remains to account for the type information in the "given" and "find" declarations. There, an expression of the form

vocabulary σ, the relation corresponding to R in a σ-structure $\mathcal{A}$ is denoted $R^{\mathcal{A}}$. For example, we write

$$\mathcal{A} := (A;\ R_1^{\mathcal{A}}, \dots R_n^{\mathcal{A}},\ f_1^{\mathcal{A}}, \dots f_m^{\mathcal{A}},\ c_1^{\mathcal{A}}, \dots c_k^{\mathcal{A}}),$$

where the R_i are relation symbols, the f_i are function symbols, and constant symbols c_i are 0-ary function symbols. A structure is finite if its universe is finite. For more background, see [4, 10].

R : rel $D_1 \times \ldots D_n$

indicates that the interpretation of R is a subset of the cross-product of the given types. Each of those sets is provided by the instance, and has a corresponding vocabulary symbol. So, the information provided is equivalent to the FO formula

$$\forall x_1, \ldots \ \forall x_n \ R(x_1, \ldots, x_n) \ \supset \ D_1(x_1) \wedge \ldots \wedge D_n(x_n).$$

Let us denote by Γ_T the set of formulas containing one formula of the form just described corresponding to each such expression in the E_{FO} specification Γ.

Now, let $\phi_\Gamma = \Gamma_C \wedge \Gamma_T$. The following is now immediate:

Proposition 1. *Let Γ be an E_{FO} specification with instance vocabulary σ and solution vocabulary ε. Then, for every finite $\sigma \cup \varepsilon$-structure $\mathcal{A}$, $\mathcal{A} \models \phi_\Gamma$ if and only if $\mathcal{A}$ is instance structure for Γ expanded with a solution.*

Here, $\mathcal{A} \models \phi$ means that ϕ is true in the structure $\mathcal{A}$. We rely on the reader believing that our first order formulas accurately express the semantics of E_{FO}. A rigorous proof would involve defining the notion of a structure satisfying an E_{FO} specification in terms of the (denotational) formal semantics of ESSENCE.

Remark 2. It is not hard to see that essentially the same logic-based formalization approach works nicely for several other languages, such as EaCL and ESRA. In the case of MXG, this formalization came before the language design [14]. Some other cases are less straightforward. For example, NPSpec semantics involve the notion of minimal Herbrand models.

3 E_{FO} Captures NP

How expressive is our fragment E_{FO}? We have the following.

Theorem 1. *Let K be a class of finite σ-structures. The following are equivalent*

1. *$K \in NP$;*
2. *There is a first-order formula ϕ with vocabulary $\sigma' \supsetneq \sigma$, such that $\mathcal{A} \in K$ if and only if there is an expansion $\mathcal{A}'$ of $\mathcal{A}$ to σ' with $\mathcal{A}' \models \phi$;*
3. *There is an E_{FO} specification Γ with instance vocabulary σ and vocabulary $\sigma' \supsetneq$ sigma, such that $\mathcal{A} \in K$ iff there is an expansion $\mathcal{A}'$ of $\mathcal{A}$ to σ' with $\mathcal{A}' \models \Gamma_T \cup \Gamma C$.*
4. *K is the class of finite models of a sentence of Existential Second Order logic ($\exists SO$), where the predicate symbols in $\sigma' - \sigma$ are existentially quantified.*

The equivalence of 1 with 2 and 4 are by Fagin's Theorem [5] (2 is just a restatement of 4).[3] The equivalence of 3 with 2 is immediate from the above.

[3] For basics of finite model theory, including a proof of Fagin's Theorem, we refer the interested reader to [10].

The theorem tells us that E_{FO} can specify all search problems whose decision versions in NP, and no other problems. Note that this is very different from NP-completeness, which tell us that exactly the problems in NP can be *polynomially reduced* to a given problem.

One reason we might be interested in knowing a language captures a complexity class is to be sure it can express all such problems. A user expecting to solve their favourite NP-complete problem would be very upset to discover they were using a modelling language that could not describe it. Another reason might be to know that it *cannot* express problems outside the given class. This is relevant to the choice of solver technology and solver design.

E_{FO} is a nearly minimal fragment of ESSENCE that captures NP. (Smaller fragments are of marginal interest. For example, one could restrict the constraint formulas to be universal formulas in CNF. It's not clear why we would study this fragment, except to carry out a proof which would be tedious otherwise.)

Remark 3. We chose to call our fragment E_{FO}, because the formulas of the "such that" part are formulas of first order logic, and the task of finding a solution to a E_{FO} specification is the model expansion problem for FO, as described in [9, 14]. The expressive power is the same as that of Existential Second Order logic ($\exists SO$) over finite structures. The difference in terminology is related to the task. When one states that "$\exists SO$ over finite structures captures NP", the task in question is model checking. Here, our interest is actually *witnessing* the second-order quantifiers.

3.1 Extending the Fragment within NP

The fragment E_{FO} is sufficient to specify a search version of any problem in NP, but no other problems. However, E_{FO} is stripped of many of the syntactic features of ESSENCE that make it attractive, even for specifying NP search problems. One may ask: How much of ESSENCE could we add to E_{FO} to improve user convenience, without formally increasing its expressive power? Here, we give some rough indications in this direction.

Simple Type and Domain Definitions We have seen that simple forms of type information can be expressed in FO formulas by limiting the range of (untyped) first order variables. Simple type definitions, however, may introduce elements that are not part of the instance, a feature not supported by our formalization as given. This limitation can be fixed by positing an infinite structure that provides an unlimited number of "reserve" elements. A FO formula corresponding to a type definition specifies the number of elements needed from this reserve.

Arithmetic E_{FO} extended with integer arithmetic still captures NP when extended with arithmetic, provided all integer domains are restricted in size. Assume, for simplicity, that all domains are of integers. We posit the infinite structure of the integers in the background, from which elements may be used at will.

Since every ESSENCE specification restricts variables to range over finite sets, we can expand E_{FO} with arithmetic without increasing expressiveness provided we restrict type and domain definitions to keep the sizes of these sets polynomial in the instance size.

4 Nested Types Lead to the Polynomial-Time Hierarchy

Here, we consider the consequences of allowing the declaration of higher-order domains, which are produced in ESSENCE by composition (nesting) of types, and quantification over these domains. We show that extending the fragment E_{FO} with quantification over variables of such types is equivalent to quantification over sets, which gives the `such that` part the power of full second order logic (over finite domains, of course).

Second-Order (SO) Logic

Recall the second order logic (SO) allows quantification over sets (of tuples) and functions. This is expressed through quantification over predicate symbols such as, e.g. $P(x_1,\ldots,x_n)$ or function symbols such as $f(x_1,\ldots,x_n)$. An example of a SO formula is

$$\exists E\ \forall f\ \forall R\ \forall x\ \forall y\ (R(x,y) \wedge f(x) = y\ \supset\ E(x)).$$

Here, E and R are SO variables ranging over sets (of tuples), f is a SO function variable, and x and y are FO variables (those ranging over simple domains). By quantifiers we mean $\forall$, $\exists$, as in first-order (FO) and SO logic. (For more background see [4].)

A Source of SO quantification in ESSENCE

We already saw that decision variables correspond to the expansion vocabulary ε. These are implicitly existentially SO quantified. An ESSENCE specification of the form

$$Q_{Set_1} \ldots Q_{Set_n} Q_{x_1 \in Set_1} \ldots Q_{x_n \in Set_n}\ \Psi,$$

where Q denotes either $\forall$ or $\exists$, and each Set_i is a variable ranging over sets (of tuples) and set variables do not appear as arguments of other predicates amounts to general second-order quantification. An example of such a specification is one having the following `such that` statements, in which D is domain defined by an expression of the form "set of D", for some finite domain D:

$$\texttt{such that}\quad \forall_{P:T} \exists_{x\in P}(R(x) \wedge \forall_{y\in P}(R(y) \implies y = x)).$$

(Every set P of elements from D contains a unique element for which property R holds). In general, we will say that a *second-order quantifier* in an ESSENCE specification is a quantifier of the form $\forall_{P:D}$ or $\exists_{P:D}$, where D is a domain whose elements are sets of tuples (or, equivalently, are relations). Thus, $\forall_{P:\text{setoftuple}(D,D)}$ and $\exists_{P:\text{rel}D\times D}$ are also second-order quantifiers.

Note that, if a symbol R is from the expansion vocabulary (i.e., R is declared in a `find` section) then there is an implicit existential SO quantifier for R at the front of the formula, and the computational task is to find a witness for that quantifier.

Obtaining the Polynomial-Time Hierarchy

Without second order quantification in E_{FO}, we can express exactly the problems in NP, also known as Σ_1^p. (Σ_0^p is the complexity class P. We are at Σ_1^p with no second order quantifiers because the expansion/find predicates are implicitly existentially second order quantified.) If we allow second order quantification, each alternation of SO quantifiers gives us a jump in the Polynomial Time Hierarchy. If the specification contains only universal ($\forall$) second order quantifiers, then we can express problems in Σ_2^p, or $\mathrm{NP}^{\mathrm{NP}}$. If the quantification pattern is $\forall\exists$, then we have Σ_3^p, or $\mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}}$, etc. This way, all Σ levels of the Polynomial Time Hierarchy (PH) are precisely captured by Essence specifications with second-order quantification. Recall that

$$PH = \bigcup_{i \in N} \Sigma_i^p$$

Let Π_i, where $i \geq 1$ denote the fragment of Essence that consists of E_{FO} extended with second-order quantification, and where the `such that` part starts with a universal set quantifier $\forall$ and has precisely i alternations of SO quantifiers (from $\forall$ to $\exists$ or back). Π_0 is FO, so the fragment of Essence corresponding to Π_0 is E_{FO}. We assume a counterpart of Definition 1 for these more expressive fragments, where Π_i replaces E_{FO}.

Theorem 2. Essence *specifications in the fragment Π_i capture Σ_{i+1}^p. That is, every Π_i-definable class of finite structures is in Σ_{i+1}^p, and every class of finite structures in Σ_{i+1}^p is Π_i-definable.*

Proof. (Sketch) It is sufficient to establish a correspondence between Essence specifications and SO logic. We already saw that Essence specifications where quantification over sets is allowed amounts to a fragment of SO logic where the outer-most quantifiers are existential, and arbitrary depth of quantifier alternations is possible. In addition, we need to show that every SO formula starting with a block of existential quantifiers can be represented by an Essence specification of this type. The capturing result will follow immediately because it holds for the corresponding fragment of SO logic. To go from SO to Essence, simply replace QP with $Q_{P:T}$, where Q is either $\forall$ or $\exists$, and let T be a suitable type, i.e., the elements of T should be sets of tuples of the appropriate arity.

5 Succinct Domains Lead to NEXP-time

In our ESSENCE fragment E_{FO}, we require that given domains be enumerated. Not coincidentally, in Fagin's theorem – and the corollaries that say that E_{FO} (and FO-MX, and NP-Spec, etc) capture NP – the domain of the instance structure is given explicitly, by enumeration, or by the size of the domain expressed in unary.

An enumerated set of size n requires at least $n \log n$ bits to describe. ESSENCE (and many the other specification and modelling languages) allow an instance domain to be given by its size. For example, in many ESSENCE specifications a positive integer n is given, and a domain of size n is then defined, using a letting statement, as in:

```
given      n  : int
letting    Nodes be new domain int(1..n)
```

which is used frequently in the examples on the ESSENCE web page.[4] In these cases, we are describing a universe of size n with about $\log n$ bits: exponentially more efficiently. We call this a *succinct* representation.

Consider the fragment of ESSENCE that consists of E_{FO} extended with the ability to give domains succinctly. This fragment can express problems of exponentially higher complexity.

Theorem 3. *The is an NEXP-complete class K of finite structures that can be expressed by the fragment of* ESSENCE *consisting of E extended with succinct domain specification.*

Proof. The following Tiling problem satisfies the claim. We are given a set of square tile types $T = \{t_0, \ldots, t_k\}$, together with two relations $H, V \subseteq T \times T$ (the horizontal and vertical compatibility relations, respectively). We are also given an integer n in binary. An $n \times n$ tiling is a function $f : \{1, \ldots, n\} \times \{1, \ldots, n\} \mapsto T$ such that

1. $f(1, 1) = t_0$, and
2. for all $i < n$ and $j \leq n$ $(f(i, j), f(i + 1, j)) \in H$, and
3. for all $i \leq n$ and $j < n$ $(f(i, j), f(i, j + 1)) \in V$.

The problem of deciding, given T, H, V and n, whether an $n \times n$ tiling exists is NEXP-complete. (This is Problem 20.2.10 (a) in Papadimitriou's complexity text [15].) To prove our theorem, we need only provide an ESSENCE specification for the tiling problem, which is given in Figure 2.

NEXP is known to properly contain NP, and thus this problem is provably not in NP, and not expressible in E_{FO} without succinct domain representation.

[4] See http://www.cs.yorku.ac.uk/aig/constraints/AutoModel

```
given       $n$: int, Tiles enum(...)
given       $t$: Tiles
given       H: rel Tiles $\times$ Tiles, V: rel Tiles $\times$ Tiles
letting     Index be new domain int$(1 \ldots n)$
find        f : Index $\times$ Index $\rightarrow$ (total) Tiles
such that   $f(1,1) = t \ \wedge$
            $\forall_{i:Index}.\ \forall_{j:Index}.\ [i < n \Rightarrow H(f(i,j), f(i+1,j))] \wedge$
            $\forall_{i:Index}.\ \forall_{j:Index}.\ [j < n \Rightarrow V(f(j), f(i,j+1))]$
```

Fig. 2. An ESSENCE specification of a Tiling problem.

Remark 4. A key point here is that T can be quite small relative to n. To construct a solution, we have to build the function f, which is of size at least n^2. This may be exponentially larger than the input, which may be of size only $O(logn)$. The fact that the problem is NEXP-complete tells us that, even if we choose a "compact representation scheme" for the function f, under any such scheme there will be values of n for which the representation of f is still of exponential size. We could even reformulate the problem as a decision problem, and just output a single yes/no bit, but this would still not significantly reduce the work to find the solution.

Should We Worry?

The above proof can be adapted to several of the languages mentioned above, including the languages for Spec2SAT [2] (an NPSpec solver), MXG [14], ASPPS [3], and other tools targeted specifically toward NP-search problems. The reason is simple: all provide a means of giving as part of an instance a range of values, as in $1..n$, as an abbreviation of the enumerated set $\{1, \ldots n\}$. This shows that they may have much greater expressive power than anticipated.

Should be be concerned? These tools have been designed primarily to solve NP-search problems, and in many respects do an admirable job. There is a simple reason that, most of the time, we should ignore this complexity result. Consider, for example, a problem where the input is a graph $G = \langle V, E \rangle$. If, by convention, we number the vertices of each n-vertex graph $\{1, \ldots n\}$, requiring the user to enumerate the n numbers explicitly rather than give the range is an inconvenience, but typically of no significance computationally. The reason is that the "hardness" is captured in the set of edges, which will be given explicitly, and has size $\Omega(n)$ in almost every application. From this point of view, nothing is amiss, but the formal properties do not exactly match "normal use".

Yet, it is possible that users could be "bitten", unexpectedly, by the property, and we think it is important that users understand the power of their tools, and under what conditions they behave "normally", or otherwise.

6 Large Defined Domains

ESSENCE allows defining new domains as a function of what is given, and these domains can be extremely large — much larger than the single exponential that appears in the previous section. Here we give two examples. We will not assume the input may be given succinctly, but rather use other features of the language.

Case 1: Composition of Type Constructors

Suppose we are given an instance domain D of size n. Then, using ESSENCE type constructors and `letting` statements, we may construct domains as follows

- set of D, which is of size 2^n;
- partition of D, which is of size 2^n;
- rel of $D \times D \times \ldots D$, which is of size 2^{n^k}, where k is the arity;
- function: $D^k \mapsto D$, which is of size 2^{n^k};
- etc.

Since these can be composed, a `letting` statement of size k can define a domain of size

$$2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}$$

where the stack of 2's is k high. Thus, we may specify problems where determining if there is a solution requires construction of an object of this size, and probably one where the smallest efficiently checkable witness to a solution is of this size. Since a formula in the `such that` part may have quantifier depth k, it can talk about elements at the inner-most level of such a nested type, so it seems we can specify problems whose complexity is at least $2^{2^{\cdots^{2^n}}}$, where the stack of 2's is k high, although we do not provide a concrete example here.

Case 2: Exponentiation

Since ESSENCE includes an exponentiation operator, and we can define a new domain in terms of it's size, we can define a domain of size n^n, given a domain of size n. We can also compose such definitions, so a letting statement of size k can define a domain of size $n^{n^{\cdots^n}}$, where the stack of n's is k high.

Such a problem specification would look like this:

```
given        D enum
letting      n be |D|
letting      Big be new domain of size n∗∗(n∗∗(n∗∗(...n)...)))
find         f : Big → (total) Big
such that    ...
```

Again, we do not provide a concrete example of a problem with solution complexity matching the size of this constructed domain, but we believe such a problem can be specified with ESSENCE.

Conjecture 1. ESSENCE can specify problems which are complete for the complexity class NTIME$[n^{n^{\cdot^{\cdot^{\cdot^{n}}}}}]$, where the stack of n's is k high for arbitrary $k \in N$.

Remark 5. The NEXP case is a special case of this one. It is interesting in it's own right because: 1) many languages allow compact representation of domains and, 2) it is related to a more general problem of numbers, which arises even without exponentiation.

A Complexity Upper Bound on ESSENCE Specifications

This does bring us to the only *upper bound* we are fairly confident of for complexity of problems specifiable in ESSENCE:

Conjecture 2. For every ESSENCE-specifiable problem P, there exists $k \in \mathbb{N}$ such that P is of complexity at most NTIME$[n^{n^{\cdot^{\cdot^{\cdot^{n}}}}}]$, where the stack of n's is k high.

The reason that the upper bound is a conjecture is that ESSENCE has a very large collection of features, and it is hard to be absolutely sure one has taken into account everything that can be done with them.

7 Capturing NP-search

Complexity theory primarily studies decision problems. A decision problem is formalized as the set of "yes" instances. For many applications, we are interested in search problems, that is, given an instance we want to *find* an object of a certain sort – called a solution – not just answer a yes-no question. A search problem is formalized as a binary relation R, such that the solutions for instance are the elements of $\{y : R(x, y)\}$. An NP-search problem is a polytime binary relation $R(x, y)$ such that the set $\{x : \exists\ y\ R(x, y)\}$ is in NP, and there is some polynomial p such that $R(x, y) \Rightarrow |y| \leq p(|x|)$.

Now, suppose you are faced with an NP-search problem $R(x, y)$. You have a specific notion of what constitutes an instance and a solution: This gives you an instance vocabulary σ, and a solution vocabulary ε. Theorem 1 tells us that there must be some E_{FO} specification S, with `find` vocabulary σ, which specifies the set $\{x : \exists\ y\ R(x, y)$. It does *not* tell you that there is such an S which has `find` vocabulary ε. Indeed, there are cases for which there is no such S. For these problems, the specification S has `given` vocabulary σ and `find` vocabulary $\varepsilon \cup \alpha$. Intuitively, one cannot write the `such that` conditions in a FO formula using only the `given` and `find` vocabulary – you must use additional concepts for which you require additional vocabulary symbols, which we may call "auxiliary" vocabulary.

We say a search problem $R(x, y)$ is E_{FO}-definable iff there is an E_{FO} specification S such that x is and S-instance with S-solution y iff $R(x, y)$.

Theorem 4. E_{FO}*, even when extended with the features listed at the end of Section 2.2, does not capture NP-search.*

Proof. (Sketch) E_{FO} can express the same class of NP checking relations as FO. However, FO can only express those relations that are in the complexity class AC_0 (this follows from the fact that FO model checking is in AC_0). This is known to be a proper subset of the polytime relations, and thus does not include all NP checking relations.

Consider now extending the formulas allowed in the `such that` expressions of E_{FO} as follows.

Definition 2. *An* ∃SO universal Horn formula *is a SO formula consisting of a conjunction of Horn clauses, preceded by universal first-order quantifiers, preceded by existential SO quantifiers. Here, a clause is Horn if it contains at most one positive occurrence of a SO variable ranging over sets (of tuples).*

Theorem 5. E_{FO}*, when extended with universal Horn formulas in the* `such that` *part, captures NP search over ordered structures.*

Proof. Follows from the capturing P result by ∃SO universal Horn [8].

Remark 6. The requirement for ordered structures in the theorem indicates that every domain must have a linear order. This is essential to the proof, and thus one would be restricted in the use of the so-called "un-named types" of ESSENCE.

Remark 7. An alternate way to extend E_{FO} to capture NP search would be to add a least-fixpoint operator, or other form of inductive definition. In our opinion, inductive definitions (or other forms of fixpoint operators) are an important element missing from almost all constraint modelling languages.

8 Discussion

Like the ESSENCE designers, we think a good industrial modelling languages should support a rich type system, and that an industrial worker with basic discrete math background but little training specific to constraint modelling, should be able to specify problems in such a language.[5] We also strongly believe that the languages used in practice to specify or model problems should have expressive power commensurate with the complexity of the problems being specified. This is, in our view, important to producing high-quality and high-performance tools. (As an illustrative, if extreme, example, imagine using a theorem prover to sort the relations of a large database.)

We consider development of ESSENCE and related languages an extremely important direction for the constraints field. In this paper, we have pointed out some expressiveness and complexity implications of providing rich linguistic features in ESSENCE. In particular, we have shown that a number of features of ESSENCE lead to extremely high expressive power. This does not constitute

[5] Of course, tackling the hardest problems will always require substantial modelling experience and understanding of solver technology.

a flaw in the design of ESSENCE, but for such expressive languages it may be useful to identify the largest fragments possible with limited expressive power, and to exercise care when using features that give expressive power beyond that required for the problem at hand.

We consider it a central challenge to researchers in the area to find ways to incorporate the kind of features that ESSENCE provides in such a way that expressiveness and complexity – but not naturalness – are constrained. This will require study of the ways features are used in natural specifications, over a very wide range of real application problems, as well as application of ingenuity and the application of appropriate design and analysis techniques.

References

1. M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. *Computer Languages*, pages 165–195, 2000.
2. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162:89–120, 2005.
3. Deborah East and Mirolsaw Truszczynski. Predicate-calculus based logics for modeling and solving search problems. *ACM TOCL*, 7(1):38–83, 2006.
4. H. B. Enderton. *A Mathematical Introduction to Logic.* Harcourt/Academic Press, New York, 2001.
5. R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Comput.*, pages 43–73, 1974.
6. P. Flener, J. Pearson, and M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems. In *Proc., LOPSTR'03*, 2003.
7. Alan M Frisch, Matthew Grum, Chris Jefferson, Bernadette Martinez Hernandez, and Ian Miguel. The design of ESSENCE: a constraint language for specifying combinatorial problems. In *Proc., 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, January 2007.
8. E. Grädel. Capturing Complexity Classes by Fragments of Second Order Logic. *Theoretical Computer Science*, 101:35–57, 1992.
9. Antonina Kolokolova, Yongmei Liu, David Mitchell, and Eugenia Ternovska. Complexity of expanding a finite structure and related tasks. In *Logic and Computational Complexity, workshop associated with LICS'06*, 2006.
10. L. Libkin. *Elements of Finite Model Theory.* Springer, 2004.
11. Maarten Mariën, Johan Wittocx, and Marc Denecker. The IDP framework for declarative problem solving. In *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.
12. P. Mills, E.P.K. Tsang, R. Williams, J. Ford, and J. Borrett. EaCL 1.0: an easy abstract constraint programming language. Technical Report CSM-321, University of Essex, December 1998.
13. David Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In *Proc. of the 20th National Conf. on Artif. Intell. (AAAI)*, pages 430–435, 2005.
14. David Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, School of Computing Science, Simon Fraser University, 2006.
15. Christos Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.