

Logical and complexity-theoretic aspects of models of computation with restricted access to arrays

Iain A. Stewart

Department of Computer Science, University of Durham,
Science Labs, South Road, Durham DH1 3LE, U.K.
e-mail: `i.a.stewart@durham.ac.uk`

May 17, 2008

Abstract

We study a class of program schemes, NPSB, in which, aside from basic assignments, non-deterministic guessing and while loops, we have access to arrays; but where these arrays are binary write-once in that they are initialized to ‘zero’ and can only ever be set to ‘one’. We show, amongst other results, that: NPSB can be realized as a vectorized Lindström logic; there are problems accepted by program schemes of NPSB that are not definable in the bounded-variable infinitary logic $\mathcal{L}_{\infty\omega}^b$; all problems accepted by the program schemes of NPSB have asymptotic probability 1; and on ordered structures, NPSB captures the complexity class $\mathbf{L}^{\mathbf{NP}}$. We give equivalences (on the class of all finite structures) of the complexity-theoretic question ‘Does \mathbf{NP} equal \mathbf{PSPACE} ?’ where the logics and classes of program schemes involved in the equivalent statements define or accept only problems with asymptotic probability 0 or 1 and so do not cover many computationally trivial problems. The class of program schemes NPSB is actually the union of an infinite hierarchy of classes of program schemes. Finally, when we amend the semantics of our program schemes slightly, we find that the classes of the resulting hierarchy capture the complexity classes Σ_i^p (where $i \geq 2$) of the Polynomial Hierarchy \mathbf{PH} .

keywords: finite model theory; descriptive complexity; program schemes.

1 Introduction

Finite model theory is essentially the study of logical definability over finite structures. An important sub-area of finite model theory is the relationship between the logical definability of classes of finite structures and computational complexity theory; that is, *descriptive complexity*. This relationship is best exemplified by Fagin’s seminal result that a problem, *i.e.*, a class of finite structures over the same signature, can be defined by a sentence of existential second-order logic if, and only if, the problem (or, to be more precise, an encoding of it) can be accepted by a polynomial-time non-deterministic Turing machine [14]. Not only does this ‘logical approach’ to computational complexity benefit from years of research in model theory which have provided new tools and methods not obviously available if one restricts oneself to the

Turing-based world of computation, but it has provoked a reexamination of logical questions and a development of new tools and methods specifically related to and designed for finite structures.

Logical characterizations of complexity classes in descriptive complexity tend to assume that there is an underlying built-in ordering of the data. This ordering is generally used to simulate (the tape-head movements of) an appropriate Turing machine. Perhaps the most compelling open problem in descriptive complexity is whether there exists a logic capturing \mathbf{P} on unordered data (that is, on the class of all finite structures) or whether one must always assume that the data is ordered (as is the case in, for example, the Immerman-Vardi-Sazonov characterization of \mathbf{P} as the class of problems definable in first-order logic extended with an inflationary fixed-point operator and with a built-in successor relation [22, 30, 42]).

An advantage of adopting a logical framework for the study of computational complexity is that one can delineate fundamental issues, like whether the data is ordered, which is extremely difficult in traditional complexity theory where the Turing machine model, working on strings as it does, insists that the data be ordered. Nowhere is this delineation more necessary than in database theory where the *data independence principle* means that database query languages can not assume any underlying ordering of the data. It is probably in the realm of query languages where finite model theory and database theory interact most. There are numerous paradigms for the design of database query languages including: *logic*, where the query languages are nothing other than logics, often the same logics as studied in finite model theory, such as first-order logic extended with an inflationary [8] or a least fixed-point [20] operator; *programming languages*, where there are variables taking relations as their values, where variables can take values derived from first-order combinations of existing variables, *i.e.*, relations, and where there is some mechanism for iteration, such as the *while* language [8], the *while*⁺ language [3] and the languages of [27] obtained by extending first-order logic with for-loops; and *logic programming*, where new relation variables are available, where there are rules to change the values of the new relation variables and where there is an appropriate semantics so as to explain potential anomalies in the applications of the rules, such as *Datalog*[−] with stratified semantics [9], well-founded semantics [18] and inflationary fixed-point semantics [1]. It turns out that many of the resulting query languages are equivalent. For example, the following query languages are all equivalent (on unordered data): first-order logic extended with an inflationary or a least fixed-point operator; the *while*⁺ programming language; and *Datalog*[−] with either well-founded or fixed-point semantics (on ordered data, these query languages all capture the complexity class \mathbf{P}). We refer the reader to [43] for more on this and for an excellent discussion of the general relationship between database theory and finite model theory.

Abiteboul and Vianu [4, 5] initiated an attempt to deal with the difficulty, mentioned above, that Turing machines work on ordered data and remain the mainstay of computational complexity theory yet database theory insists that all of its query languages must necessarily work on unordered data. Their work was subsequently developed in [6], in collaboration with Vardi, where the *relational machine* was defined. A relational machine works directly on an input structure (and not its encoding as a string) and has both a traditional Turing machine component and a facility to manipulate relations using relational algebra operations. Abiteboul, Vianu and Vardi used

their machines to define *relational complexity classes* and proved that their class P_r , namely relational polynomial-time, is nothing other than first-order logic extended with an inflationary fixed-point operator, and that their class $PSPACE_r$, namely relational polynomial-space, is nothing other than first-order logic extended with a partial fixed-point operator (we reiterate that these equivalences are on unordered data). Moreover, they also showed the remarkable equivalence that $P_r = PSPACE_r$ if, and only if, $\mathbf{P} = \mathbf{PSPACE}$.

In recent papers, we have developed new programming languages (for unordered data) based around the paradigm of a *program scheme*. Program schemes are more computational in flavour than are formulae of traditional logics yet they remain amenable to logical manipulation. The concept of a program scheme originates from the 1970's with work of, for example, Constable and Gries, Friedman, and Hewitt and Paterson [10, 15, 29], and complexity-theoretic considerations of such program schemes were subsequently studied by, for example, Harel and Peleg, Jones and Muchnik, and Tiuryn and Urzyczyn [21, 25, 41]. Our analysis of program schemes differs from what has gone before in that we are always concerned with finite structures (and not infinite ones as was often the case previously) and we do not assume that the elements of our finite structures are necessarily linearly ordered. Our program schemes differ from the programming languages of database theory in that variables take elements of the input structure as values and not relations. Working with such a 'low-level' computational device allows us to incorporate well-known programming constructs such as stacks, arrays, queues, non-determinism, while-loops, for-loops, and so on, in our programming languages. Furthermore, as the computations of our devices manipulate the values of variables, *i.e.*, elements of the input structure, in a precise and well-defined way, we have managed to obviate the need to use existing, more established paradigms; for example, we do not need to have any experience of bounded-variable logics and their relationship with various pebble games in order to perform our analysis.

Let us summarize some of the results we have so far established (all results mentioned below are on the class of all finite structures, except where explicitly stated otherwise). In [7], we considered a hierarchy of classes of program schemes, NPSS, where these program schemes involve assignments, while-loops and non-deterministic guessing and have access to a stack, and showed that this hierarchy is proper and (the union of it) has exactly the same expressive power as stratified fixed-point logic and stratified *Datalog* (and so on ordered structures captures \mathbf{P}). On the class of strongly-connected locally-ordered digraphs, the class of connected planar embeddings and the class of triangulations, NPSS still captures exactly the polynomial-time solvable problems [40]. If we remove access to the stack then the resulting class of program schemes, NPS, is equivalent to Immerman's transitive closure logic. In [39], we show that the class of program schemes NPS augmented with arrays, NPSA, captures the class of problems defined by the sentences of a logic formed by extending first-order logic with a particular uniform (or vectorized) sequence of Lindström quantifiers, and that this logic has a zero-one law. However, we show that there are problems definable in a basic fragment of this logic which are not definable in the bounded-variable infinitary logic $\mathcal{L}_{\infty\omega}^\omega$. As a consequence, the class of problems (accepted by the program schemes of) NPSA is not contained in the class of problems defined by the sentences of partial fixed-point logic, even though in the presence of a built-in suc-

cessor relation, both NPSA and partial fixed-point logic capture the complexity class **PSPACE**. In [38], we prove that a basic class of program schemes augmented with access to a queue, NPSQ(1), is exactly the class of recursively enumerable problems that are closed under extensions. We define an infinite hierarchy of classes of program schemes for which NPSQ(1) is the first class and the union of the classes of which is the class NPSQ. We show that the class of problems NPSQ is the union of the classes of problems defined by the sentences of all vectorized Lindström logics formed using operators whose corresponding problems are recursively enumerable and closed under extensions; as a result, every problem in this class has asymptotic probability 0 or 1. Finally, in [17], we define a (deterministic) class of program schemes RFDPS constructed around notions of forall-loops, repeat-loops and arrays. We show that the class of problems RFDPS properly contains the class of problems definable in inflationary fixed-point logic (for example, the well-known problem Parity is in RFDPS) and that there is a strict, infinite hierarchy of classes of problems within RFDPS (the union of which is RFDPS) parameterized by the depth of nesting of forall-loops in our program schemes. Suffice it to say, examining classes of program schemes formed using well understood programming constructs has resulted in new, natural ‘logics’ that seemingly have hitherto arisen in neither in finite model theory nor database theory.

In this paper, in an attempt to ‘bridge the gap’ between the polynomial-time world of NPSS and the polynomial-space world of NPSA, we modify the program schemes of NPSA so that all arrays are ‘binary write-once’ in the sense that all array elements are initially set at ‘zero’ and the only modification to any array element allowed is to set it to ‘one’. The resulting class of program schemes is denoted NPSB, with the levels of the underlying hierarchy being NPSB(i), for $i \geq 1$. We show that NPSB retains some of the properties of NPSA: like NPSA, NPSB can be realized as a vectorized Lindström logic, and NPSB(1) contains problems not definable in $\mathcal{L}_{\infty\omega}^{\omega}$ (every problem accepted by a program scheme of NPSB trivially has asymptotic probability 0 or 1 as NPSB is a sub-class of NPSA); but whereas both NPSA and NPSA(1) capture **PSPACE** on ordered structures, NPSB(1) captures **NP** and NPSB captures **L^{NP}**, with the NPSB hierarchy collapsing to NPSB(3). We compare the relative expressibilities of the classes of program schemes NPSA(1) and NPSB(1). We show that $\text{NPSA}(1) = \mathbf{PSPACE} \cap \mathcal{EXT}$ and $\text{NPSB}(1) = \mathbf{NP} \cap \mathcal{EXT}$, where \mathcal{EXT} is the class of problems closed under extensions (and hence every non-trivial problem in \mathcal{EXT} has asymptotic probability 1); thus, the question of whether **NP** is equal to **PSPACE** is equivalent to the question of whether the two classes of program schemes NPSB(1) and NPSA(1) accept the same class of problems on the class of all finite structures (of course, NPSB(1) and NPSA(1) also coincide with fragments of the Lindström logics mentioned above).

Finally, we amend the semantics of the class of program schemes NPSB in that we allow the current values of arrays to be ‘passed across’ to other program schemes (appearing as, what amounts to, subroutines in the main program scheme) in a computation (hitherto, the semantics has only allowed the current values of variables to be passed across). We denote the class of program schemes with this amended semantics as NPSB^p (the subscript reflects the polynomially many values passed across to the component program schemes). We show that on the class of all finite structures, $\text{NPSB}^p(2i - 1) = \text{NPSB}^p(2i)$ and captures Σ_i^p , *i.e.*, the i th level of the Polynomial

Hierarchy **PH**, for $i \geq 2$, and so NPSB^p captures **PH** itself.

This paper is structured as follows. In the next section, we outline the definitions relating to this paper before we define our classes of program schemes in Section 3. In Section 4, we identify the class of program schemes NPSB with a vectorized Lindström logic. In Section 5, we consider our program schemes and logics on ordered structures and compare the relative computational power of the classes of program schemes $\text{NPSB}(1)$ and $\text{NPSA}(1)$ (on the class of all finite structures). We amend the semantics of our program schemes in Section 6 before presenting our conclusions and directions for further research in Section 7.

2 Preliminaries and program schemes

The main reference texts for the basic concepts, notions and results of finite model theory are [12, 24, 26] and it is to these books that we refer the reader for such information. Ordinarily, a *signature* σ is a tuple $\langle R_1, \dots, R_r, C_1, \dots, C_c \rangle$, where each R_i is a relation symbol, of arity a_i , and each C_j is a constant symbol. *First-order logic over the signature* σ , $\text{FO}(\sigma)$, consists of those formulae built from atomic formulae over σ using $\wedge, \vee, \neg, \forall$ and \exists ; and $\text{FO} = \cup\{\text{FO}(\sigma) : \sigma \text{ is some signature}\}$.

A *finite structure* \mathcal{A} over the signature σ , or σ -*structure*, consists of a finite *universe* or *domain* $|\mathcal{A}|$ together with a relation R_i of arity a_i , for every relation symbol R_i of σ , and a constant $C_j \in |\mathcal{A}|$, for every constant symbol C_j (by an abuse of notation, we do not usually distinguish between constants and relations, $C_j^{\mathcal{A}}$ and $R_i^{\mathcal{A}}$, and constant and relation symbols, C_j and R_i). A finite structure \mathcal{A} whose domain consists of n distinct elements has *size* n , and we denote the size of \mathcal{A} by $|\mathcal{A}|$ also (this does not cause confusion). We only ever consider finite structures of size at least 2, and the set of all finite structures of size at least 2 over the signature σ is denoted $\text{STRUCT}(\sigma)$. A *problem* over some signature σ consists of a subset of $\text{STRUCT}(\sigma)$ that is closed under isomorphism; that is, if \mathcal{A} is in the problem then so is every isomorphic copy of \mathcal{A} . Throughout, all our structures are finite.

The class of problems defined by the sentences of FO is denoted by FO also, and we do likewise for other logics. It is widely acknowledged that, as a means for defining problems, first-order logic leaves a lot to be desired; especially when we have in mind developing a relationship between computational complexity and logical definability. For example, every first-order definable problem can be accepted by a log-space deterministic Turing machine (where structures are encoded as strings) yet there are problems in the complexity class **L** (log-space) which can not be defined in first-order logic (one such being the problem consisting of all those structures, over any signature, that have even size). Consequently, a number of methods have been developed so as to increase definability.

One method is to extend first-order logic using a *vectorized sequence of Lindström quantifiers* corresponding to some problem Ω ; or, as we prefer, an *operator* Ω for short. Suppose that Ω is over the signature σ , where $\sigma = \langle R_1, \dots, R_r, C_1, \dots, C_c \rangle$, as above. The logic $(\pm\Omega)^*[\text{FO}]$ consists of those formulae built using the usual constructs of first-order logic and also the operator Ω , where the operator Ω is applied as follows.

- Suppose that $\psi_1(\mathbf{x}_1, \mathbf{y}), \dots, \psi_r(\mathbf{x}_r, \mathbf{y})$ are formulae of $(\pm\Omega)^*[\text{FO}]$ such that:
 - each \mathbf{x}_i is a ka_i -tuple of distinct variables, for some fixed $k \geq 1$;

- \mathbf{y} is an m -tuple of distinct variables, for some $m \geq 0$, each of which is different from any variable of $\mathbf{x}_1, \dots, \mathbf{x}_r$; and
- all free variables of any ψ_i are contained in either \mathbf{x}_i or \mathbf{y} .
- Suppose that $\mathbf{d}_1, \dots, \mathbf{d}_c$ are k -tuples of variables and constants (which need not be distinct).
- Then:

$$\Omega[\lambda \mathbf{x}_1 \psi_1(\mathbf{x}_1, \mathbf{y}), \dots, \mathbf{x}_r \psi_r(\mathbf{x}_r, \mathbf{y})](\mathbf{d}_1, \dots, \mathbf{d}_c)$$

is a formula of $(\pm\Omega)^*[\text{FO}]$ whose free variables are the variables of \mathbf{y} together with any other variables appearing in $\mathbf{d}_1, \dots, \mathbf{d}_c$.

If Φ is a sentence of the form $\Omega[\lambda \mathbf{x}_1 \psi_1(\mathbf{x}_1), \dots, \mathbf{x}_r \psi_r(\mathbf{x}_r)](\mathbf{d}_1, \dots, \mathbf{d}_c)$, as above, over some signature σ' then we interpret Φ in a σ' -structure \mathcal{A} as follows (note that as Φ is a sentence, the variables of \mathbf{y} are absent and the tuples $\mathbf{d}_1, \dots, \mathbf{d}_c$, which are only there if there are constant symbols in σ , consist entirely of constant symbols of σ').

- The domain of the σ -structure $\Phi(\mathcal{A})$ is $|\mathcal{A}|^k$.
- The relation R_i of $\Phi(\mathcal{A})$ is defined via:
 - for any $\mathbf{u} \in |\Phi(\mathcal{A})|^{a_i} = |\mathcal{A}|^{ka_i}$, $R_i(\mathbf{u})$ holds in $\Phi(\mathcal{A})$ if, and only if, $\psi_i(\mathbf{u})$ holds in \mathcal{A} .
- The constant C_j of $\Phi(\mathcal{A})$ is defined via:
 - C_j is the interpretation of the tuple of constants \mathbf{d}_j in \mathcal{A} .

We define that $\mathcal{A} \models \Phi$ if, and only if, $\Phi(\mathcal{A}) \in \Omega$ (the situation where Φ has free variables is similar except that Φ is interpreted in expansions of σ' -structures by an appropriate number of constants). We call logics such as $(\pm\Omega)^*[\text{FO}]$ *vectorized Lindström logics*. We shall also be interested in fragments of vectorized Lindström logics where: the formulae are such that the operator Ω does not appear within the scope of a negation sign, namely $\Omega^*[\text{FO}]$ (the *positive* fragment of $(\pm\Omega)^*[\text{FO}]$); and further the formulae are such that there are no nestings of the operator Ω , namely $\Omega^1[\text{FO}]$ (the *positive unnested* fragment of $(\pm\Omega)^*[\text{FO}]$).

It can be the case that (a fragment of) a vectorized Lindström logic $(\pm\Omega)^*[\text{FO}]$ has a very straightforward normal form; a normal form which obviates the need to nest applications of the operator Ω and which tells us something about the ‘degree of difficulty’ of the particular problem Ω with respect to the class of problems defined by the sentences of the logic. For example, suppose that every problem in (a fragment of) $(\pm\Omega)^*[\text{FO}]$ can be defined by a sentence of the form $\Omega[\lambda \mathbf{x}_1 \psi_1(\mathbf{x}_1), \dots, \mathbf{x}_r \psi_r(\mathbf{x}_r)](\mathbf{d}_1, \dots, \mathbf{d}_c)$, as above, except where each ψ_i is quantifier-free first-order. Then we say that the problem Ω is *complete* for (the fragment of) $(\pm\Omega)^*[\text{FO}]$ via *quantifier-free first-order translations*. This is directly analogous to completeness for some complexity class via some resource-bounded reduction; in fact, as we shall see, such normal forms can often yield very strong complexity-theoretic completeness results.

Vectorized Lindström logics have been studied quite extensively in finite model theory and a whole range of complexity classes have been *captured*, *i.e.*, characterized, by vectorized Lindström logics (see, for example, [19, 23, 31, 32] and the references

therein). However, some (though not all) of these characterizations only hold in the presence of a built-in successor relation. Consider some vectorized Lindström logic $(\pm\Omega)^*[\text{FO}]$. To say that this logic has a *built-in successor relation*, which we denote by $(\pm\Omega)^*[\text{FO}_s]$, means that no matter which signature σ' we are working over, there is always a binary relation symbol *succ* and two constant symbols 0 and *max* available (none of which is in σ') such that *succ* is always interpreted as a successor relation with least element 0 and greatest element *max* in any σ' -structure. That is, for any σ' -structure of size n , *succ* is always of the form $\{(0, u_1), (u_1, u_2), \dots, (u_{n-2}, \text{max})\}$, where the elements of $\{0, u_1, u_2, \dots, u_{n-2}, \text{max}\}$ are distinct. However, there is a further semantic stipulation on the sentences of $(\pm\Omega)^*[\text{FO}_s]$: we only consider as well-formed those sentences for which the interpretation in any structure is independent of the particular successor relation chosen. For example, define the problem TC over the signature $\sigma_{2++} = \langle E, C, D \rangle$, where E is a binary relation symbol and C and D are constant symbols, as consisting of all those σ_{2++} -structures for which, when considered as digraphs in the natural way, there is a directed path from the vertex C to the vertex D . Then the following sentence is a well-formed sentence of $(\pm\text{TC})^*[\text{FO}_s]$ (as satisfiability in a given structure is invariant with respect to *succ*):

$$\begin{aligned} & \text{TC}[\lambda(x_1, x_2), (y_1, y_2)(x_1 = 0 \wedge y_1 = \text{max} \wedge \text{succ}(x_2, y_2)) \\ & \quad \vee (x_1 = \text{max} \wedge y_1 = 0 \wedge \text{succ}(x_2, y_2))](0, 0, \text{max}, \text{max}), \end{aligned}$$

and it defines the problem over the empty signature consisting of those structures of even size. (Note that in [12], the mechanism by which a successor relation is introduced into a logic is slightly different from how we have described here in that only problems on *ordered structures* are ever considered: see [12]. Nevertheless, the two approaches essentially amount to the same thing and we shall refer to the two mechanisms interchangeably. Note also that other relations can be built into logics in the same way as is a successor relation; or even just two distinct constants can be built in.)

From a logical perspective, there is a problem with our built-in successor relation in the following sense. Given a sentence of first-order logic in which the relation symbol *succ* appears (and in which other constant and relation symbols might appear), it is actually undecidable as to whether the sentence is invariant with respect to *succ*. That is, there does not exist an effective enumeration of the well-formed sentences of FO_s . Given this fact, it is highly debatable as to whether any ‘logic’ $(\pm\Omega)^*[\text{FO}_s]$ should really be called a logic; and it is an open question currently occupying much research activity as to whether there actually exists a logic capturing the complexity class \mathbf{P} (polynomial-time), or indeed any complexity class contained in \mathbf{NP} (non-deterministic polynomial-time), where by ‘contained in’ we really mean ‘contained in but expected to be different from’ (as \mathbf{NP} itself can be captured by a logic, one such being existential second-order logic). The reader is referred to [28] for more details on this and related points. (Notwithstanding the above discussion, we still refer to $(\pm\Omega)^*[\text{FO}_s]$ as a logic on the grounds of convenience.)

Theorem 1, below, is an example of a normal form result. Define the problem CUB, over the signature $\sigma_2 = \langle E \rangle$, where E is a binary relation symbol, as follows.

$$\text{CUB} = \{\mathcal{G} \in \text{STRUCT}(\sigma_2) \quad : \quad \text{the graph } \mathcal{G} \text{ has a subset of edges inducing a regular subgraph of degree 3}\}$$

(think of a σ_2 -structure \mathcal{G} as encoding an undirected graph via: ‘there is an edge (u, v) if, and only if, $u \neq v$ and $E(u, v) \vee E(v, u)$ holds in \mathcal{G} ’). We shall need the following result later on.

Theorem 1 [36] *The complexity class \mathbf{NP} is identical to the class of problems defined by the sentences of $CUB^*[FO_s]$; and any problem in \mathbf{NP} can be defined by a sentence of $CUB^1[FO_s]$ of the form:*

$$CUB[\lambda \mathbf{x}, \mathbf{y} \psi(\mathbf{x}, \mathbf{y})],$$

where $|\mathbf{x}| = |\mathbf{y}| = k$, for some $k \geq 1$, and ψ is a quantifier-free formula of FO_s . Hence, $CUB^*[FO_s] = CUB^1[FO_s] = \mathbf{NP}$ and CUB is complete for \mathbf{NP} via quantifier-free first-order translations with successor. \square

Note that Theorem 1 subsumes the ‘traditional’ known complexity-theoretic result that CUB is complete for \mathbf{NP} via log-space reductions (a result attributed to Chvátal in [16]).

3 Program schemes

Program schemes are more ‘computational’ means for defining classes of problems than are logical formulae. A *program scheme* $\rho \in \text{NPSA}(1)$ involves a finite set $\{x_1, x_2, \dots, x_k\}$ of *variables*, for some $k \geq 1$, and is over a signature σ . It consists of a finite sequence of *instructions* where each instruction, apart from the first and the last, is one of the following:

- an *assignment instruction* of the form ‘ $x_i := y$ ’, where $i \in \{1, 2, \dots, k\}$ and where y is a variable from $\{x_1, x_2, \dots, x_k\}$, a constant symbol of σ or one of the special constant symbols 0 and *max* which do not appear in any signature;
- an *assignment instruction* of the form ‘ $x_i := A[y_1, y_2, \dots, y_d]$ ’ or ‘ $A[y_1, y_2, \dots, y_d] := y_0$ ’, for some $i \in \{1, 2, \dots, k\}$, where each y_j is a variable from $\{x_1, x_2, \dots, x_k\}$, a constant symbol of σ or one of the special constant symbols 0 and *max* which do not appear in any signature, and where A is an array symbol of dimension d ;
- a *guess instruction* of the form ‘**guess** x_i ’, where $i \in \{1, 2, \dots, k\}$; or
- a *while instruction* of the form ‘**while** φ **do** $\alpha_1; \alpha_2; \dots; \alpha_q$ **od**’, where φ is a quantifier-free formula of $\text{FO}(\sigma \cup \{0, \text{max}\})$ whose free variables are from $\{x_1, x_2, \dots, x_k\}$, and where each of $\alpha_1, \alpha_2, \dots, \alpha_q$ is another instruction of one of the forms given here (note that there may be nested while instructions).

The first instruction of ρ is ‘**input** (x_1, x_2, \dots, x_l) ’ and the last instruction is ‘**output** (x_1, x_2, \dots, x_l) ’, for some l where $1 \leq l \leq k$. The variables x_1, x_2, \dots, x_l are the *input-output variables* of ρ , the variables $x_{l+1}, x_{l+2}, \dots, x_k$ are the *free variables* of ρ and, further, any free variable of ρ never appears on the left-hand side of an assignment instruction nor in a guess instruction. Essentially, free variables appear in ρ as if they were constant symbols.

A program scheme $\rho \in \text{NPSA}(1)$ over σ with t free variables, say, takes a σ -structure \mathcal{A} and t additional values from $|\mathcal{A}|$, one for each free variable of ρ , as input;

that is, an expansion \mathcal{A}' of \mathcal{A} by adjoining t additional constants. The program scheme ρ computes on \mathcal{A}' in the obvious way except that:

- execution of the instruction ‘`guess x_i` ’ non-deterministically assigns an element of $|\mathcal{A}|$ to the variable x_i ;
- the constants 0 and max are interpreted as two arbitrary but distinct elements of $|\mathcal{A}|$; and
- initially, every input-output variable and every array element is assumed to have the value 0.

Note that throughout a computation of ρ , the value of any free variable does not change. The expansion \mathcal{A}' of the structure \mathcal{A} is *accepted* by ρ , and we write $\mathcal{A}' \models \rho$, if, and only if, there exists a computation of ρ on this expansion such that the output-instruction is reached with all input-output variables having the value max (in particular, some computations might not be terminating). We can easily build the usual ‘if’ and ‘if-then-else’ instructions using while instructions (see, for example, [33]). Henceforth, we shall assume that these instructions are at our disposal.

We want the sets of structures accepted by our program schemes to be problems, *i.e.*, closed under isomorphism, and so we only ever consider program schemes ρ where a structure is accepted by ρ when 0 and max are given two distinct values from the universe of the structure if, and only if, it is accepted no matter which pair of distinct values is chosen for 0 and max . This is analogous to how we build two constant symbols into a logic. Furthermore, we can build a successor relation into the program schemes of NPSA(1) so as to obtain the class of program schemes $NPSA_s(1)$. As with our logics, we write $NPSA(1)$ and $NPSA_s(1)$ to also denote the class of problems accepted by the program schemes of $NPSA(1)$ and $NPSA_s(1)$, respectively (and do likewise with other classes of program schemes).

We have two remarks. First, our notation $NPSA(1)$ reflects the fact that $NPSA(1)$ is the first level of an infinite hierarchy of classes of program schemes, as we shall see presently. Second, as the definition of our class of program schemes $NPSA(1)$ stands, we do not know whether the program schemes in this class can be recursively enumerated. However, we are prepared to live with this (possible) inconvenience for the following reasons. We could amend our definition of a program scheme of $NPSA(1)$ and dispense with the constant symbols 0 and max but insist that the syntax was such that the first instructions are always `guess x_0` ; `guess x_{max}` ; `while $x_0 = x_{max}$ do guess x_0 od`. Consequently, every program scheme begins by assigning two distinct values from the domain of the input structure to the variables x_0 and x_{max} . These values would then be used as the constants 0 and max , and acceptance would be signalled by any execution reaching the instruction `output`. With this amended syntax and definition of acceptance, the class of structures accepted by any program scheme is trivially closed under isomorphism and the class of program schemes is recursively enumerable; moreover, all the results in this paper would apply with our class of program schemes $NPSA(1)$ so defined (as readers can easily verify as they read through the forthcoming proofs). The essential reason for this is that all our constructions are such that they never use any ‘semantic’ information relating to 0 and max , just the fact that they are distinct elements of the input structure. We shall remark upon forthcoming definitions and results in relation to this amended

definition of NPSA(1) as we proceed. However, there are four real reasons for having the constant symbols 0 and *max* in our program schemes, with all of the reasons more to do with clarity and pragmatism than anything else. First, we can use 0 to cleanly initialize all variables and arrays, with the result that we never have to worry about whether an assignment involves an uninitialized variable or array element. Second, having two distinct constants around is useful when it comes to programming. Third, we shall soon use the constant symbol *max* to enable us to study ‘binary write-once’ arrays (that is, arrays where the elements can only be set to *max* and thereafter remain unchanged). Fourth, were we to include a built-in successor relation then having the constant symbols 0 and *max* available enables us to do this in a consistent fashion.

Henceforth, we think of our program schemes as being written in the style of a computer program. That is, each instruction is written on one line and while instructions (and, similarly, if and if-then-else instructions) are split so that ‘**while** φ **do**’ appears on one line, ‘ α_1 ’ appears on the next, ‘ α_2 ’ on the next, and so on (of course, if any α_i is a while, if or if-then-else instruction then it is split over a number of lines in the same way). The instructions are labelled 1, 2, and so on, according to the line they appear on. In particular, every instruction is considered to be an assignment, a guess or a test. An *instantaneous description (ID)* of a program scheme on some input consists of a value for each variable, the number of the instruction about to be executed and values for all array elements. A *partial ID* consists of just a value for each variable and the number of the instruction about to be executed. One *step* in a program scheme computation is the execution of one instruction, which takes one ID to another, and we say that a program scheme can *move* from one ID to another if there exists a sequence of steps taking the former ID to the latter.

As we hinted at above, the class of program schemes NPSA(1) is but the first level of an infinite hierarchy of program schemes. Suppose that we have defined a class of program schemes NPSA($2m - 1$), for some $m \geq 1$, and that any program scheme has associated with it: a set of input-output variables; a set of free variables; and a set of bound variables (this is certainly the case when $m = 1$, where the associated set of bound variables is empty).

Definition 2 Let the program scheme $\rho \in \text{NPSA}(2m - 1)$ be over the signature σ . Suppose that ρ has: input-output variables x_1, x_2, \dots, x_k ; free variables $x_{k+1}, x_{k+2}, \dots, x_{k+s}$; and bound variables $x_{k+s+1}, x_{k+s+2}, \dots, x_{k+s+t}$. Let $x_{i_1}, x_{i_2}, \dots, x_{i_p}$ be free variables of ρ , for some p (and so $k + 1 \leq i_1 < i_2 < \dots < i_p \leq k + s$). Then:

$$\forall x_{i_1} \forall x_{i_2} \dots \forall x_{i_p} \rho$$

is a program scheme of NPSA($2m$), which we denote by ρ' , with: no input-output variables; free variables those of $\{x_{k+1}, x_{k+2}, \dots, x_{k+s}\} \setminus \{x_{i_1}, x_{i_2}, \dots, x_{i_p}\}$; and the remaining variables of $\{x_1, x_2, \dots, x_{k+s+t}\}$ as its bound variables.

A program scheme such as ρ' takes expansions \mathcal{A}' of σ -structures \mathcal{A} by adjoining $s - p$ constants as input (one for each free variable), and ρ' accepts such an expansion \mathcal{A}' if, and only if, for every expansion \mathcal{A}'' of \mathcal{A}' by p additional constants (one for each variable x_{i_j} , for $j \in \{1, 2, \dots, p\}$), $\mathcal{A}'' \models \rho$ (the computation on such an expansion \mathcal{A}'' always starts with the arrays initialised to 0). \square

Definition 3 A program scheme $\rho' \in \text{NPSA}(2m - 1)$, for some $m \geq 2$, over the signature σ , is defined exactly as is a program scheme of $\text{NPSA}(1)$ except that the test in any while instruction is a program scheme $\rho \in \text{NPSA}(2m - 2)$. The bound variables of ρ' consist of the bound variables of any test in any while instruction; all free variables in any test in any while instruction are input-output or free variables of ρ' ; and there may be other free and input-output variables (appearing in ρ' at the ‘top level’ but not in any test). Of course, any free variable never appears on the left-hand side of an assignment instruction or in a guess instruction (at the ‘top level’).

Suppose that a program scheme $\rho' \in \text{NPSA}(2m - 1)$ has s free variables. Then it takes an expansion \mathcal{A}' of a σ -structure \mathcal{A} by adjoining s constants as input and computes on \mathcal{A}' in the obvious way; except that when some while instruction is encountered, the test, which is a program scheme $\rho \in \text{NPSA}(2m - 2)$, is evaluated according to the expansion of \mathcal{A}' by the current values of any relevant input-output variables of ρ' (which may be free in ρ). In order to evaluate this test, the arrays associated with ρ are initialized at 0 and when the test has been evaluated the computation of ρ' resumes accordingly with the values of its arrays and input-output and free variables being exactly as they were immediately prior to the test being evaluated. In particular, array values can not be ‘passed across’ in the evaluation of tests: the values of variables can be but they are never amended in the process. \square

Consequently, we obtain a hierarchy of classes of problems:

$$\text{NPSA}(1) \subseteq \text{NPSA}(2) \subseteq \dots \subseteq \cup\{\text{NPSA}(i) : i = 1, 2, \dots\} = \text{NPSA}$$

(we use the inclusion relation between consecutive classes because this is how they are related as classes of problems). It is easy to see that, for one thing, $\text{FO} \subseteq \text{NPSA}$.

In this paper, we are primarily interested in some sub-classes of program schemes of NPSA , namely the sub-classes $\text{NPSB}(i)$, for $i = 1, 2, \dots$, and the union of these classes NPSB , where the only allowed assignment instructions with an array element on the left-hand side are of the form $A[x_1, x_2, \dots, x_k] := \text{max}$; that is, the only values array elements can have are 0 and max , and once an array element is set to max then it remains at max thereafter (the notation reflects the binary nature of these arrays). Obviously, $\text{NPSB}(i) \subseteq \text{NPSA}(i)$, for all $i = 1, 2, \dots$; and $\text{NPSB} \subseteq \text{NPSA}$.

Remark 4 We note that were we to define our program schemes of $\text{NPSA}(1)$ without using the constant symbols 0 and max , as hinted at previously, then we could adapt Definitions 2 and 3 accordingly by never quantifying the variables x_0 and x_{max} . Also, the program schemes of NPSB would be defined by x_0 and x_{max} replacing 0 and max .

Results concerning the program schemes of NPSA have already been obtained, and some of these results relevant to this paper are stated below. A problem Ω , over some signature σ and where the domain of any σ -structure of size n is taken to be $\{1, 2, \dots, n\}$, for which the function $f(n)$, defined as the number of structures in Ω of size n divided by the number of σ -structures of size n , is such that the limit as n tends to infinity exists and is r is said to have *asymptotic probability* r . For us, r will always be 0 or 1. We say that a logic has a *zero-one law* if every problem definable in that logic has asymptotic probability 0 or 1.

Theorem 5 [39]

(i) *There exists a problem Ω_a , involving reachability in Petri nets, for which*

$$NPSA = (\pm\Omega_a)^*[FO],$$

and every problem in NPSA has asymptotic probability 0 or 1.

- (ii) *There is a quantifier-free first-order translation with 2 constants from any problem in NPSA(1) to the problem Ω_a ; and so Ω_a is complete for NPSA(1) via quantifier-free first-order translations with 2 constants.*
- (iii) *The problem CUB is in NPSA(1) but not definable in the logic $\mathcal{L}_{\infty\omega}^\omega$.*
- (iv) *In the presence of a built-in successor relation, the hierarchy $NPSA_s$ collapses to the first level, $NPSA_s(1)$, and captures the complexity class **PSPACE**. \square*

It is worth mentioning the role of the logic $\mathcal{L}_{\infty\omega}^\omega$ in finite model theory. This logic is an important logic for a number of reasons, one of which is that it subsumes many of the logics from finite model theory (including transitive-closure logic, least fixed point logic and partial fixed point logic) in that these logics can be realized as fragments of $\mathcal{L}_{\infty\omega}^\omega$. Furthermore, $\mathcal{L}_{\infty\omega}^\omega$ has a zero-one law and so any logic subsumed by $\mathcal{L}_{\infty\omega}^\omega$ has a zero-one law. It is particularly interesting that NPSA(1) (and so also NPSA) can not be realized as a fragment of $\mathcal{L}_{\infty\omega}^\omega$ (as CUB is a problem in NPSA(1) that is not in $\mathcal{L}_{\infty\omega}^\omega$, a result proven in [37]).

In the absence of arrays, when the resulting class of program schemes is denoted NPS, and additionally in the presence of a stack, when the resulting class of program schemes is denoted NPSS, there are results analogous to parts (i), (ii) and (iv) of Theorem 5 (see [7]) in that: both NPS and NPSS can be realized as vectorized Lindström logics so that the problems corresponding to the operators involved in these logics are complete for NPS(1) and NPSS(1) via quantifier-free first-order translations with 2 constants; and on ordered structures, the complexity classes captured are **NL** (non-deterministic log-space) and **P**, respectively. However, unlike NPSA, both NPS and NPSS can be realized as fragments of $\mathcal{L}_{\infty\omega}^\omega$. Furthermore, the underlying hierarchies of NPS and NPSS are proper at every level (even if we restrict to problems only involving trees) whereas, as we shall affirm later, all that is known as regards NPSA is that $NPSA(1) \subset NPSA(2) \subset NPSA(3)$.

4 Partitioned Petri nets

We begin by describing a generalization of the digraph reachability problem to a scenario where the moves between nodes depend upon the availability and utilization of external resources. We first describe the basic decision problem in an everyday fashion before we consider a manifestation of it as a class of structures over a given signature and see how this problem is related to computation in the program schemes of NPSB(1).

Consider the following scenario. We are given a directed graph $G = (V, E)$, where $|V| = n$, with a source vertex *source* and a sink vertex *sink*, but where each edge is labelled with a (possibly empty) set of labels with each label being of one of the following forms:

- ‘user resource r_i is unused’;
- ‘system resource s_j is available’;
- ‘user resource r_i is unused and this move uses this resource but makes the system resource s_j available (if it wasn’t available previously)’.

There is a polynomial number of different *user resources* $\{r_i : i = 1, 2, \dots, p(n)\}$, which are either in the state ‘used’ or the state ‘unused’; and a polynomial number of *system resources* $\{s_i : i = 1, 2, \dots, q(n)\}$, which are either in the state ‘available’ or the state ‘unavailable’ (for some polynomials p and q). A move in the digraph from vertex u to vertex v via the edge (u, v) can only be made if either no labels label the edge (u, v) or at least one of the labels labelling the edge (u, v) is satisfied (with a resulting change in the state of a user resource, and possibly a system resource, if the label is of the third type). The question we ask is, given the initial state where all user resources are unused and no system resources are available, is it possible to move from *source* to *sink* in our given environment? That is, can the user use his or her resources wisely so as to enable a traversal in the digraph from the source to the sink?

Note that whether a move can be made depends only on certain predicates involving the states of the resources: for example, there are no moves dependent upon the state of a user resource being ‘used’ or of a system resource being ‘unavailable’. The situation is as it is as this decision problem arises naturally from our consideration of our program schemes; but we comment further on this problem and related problems in the Conclusion.

We encode the above decision problem as a problem, *i.e.*, class of finite structures, involving Petri nets. Our encoding is natural and has certain properties which we shall utilize later. The reader is referred to [13] for the basic notions and concepts relating to Petri nets (this reference also gives details of numerous complexity-theoretic results concerning fundamental problems in Petri nets).

Definition 6 Define $\sigma_b = \langle P, Q, T_1, T_2, T_3, C, D \rangle$ where P , Q , T_1 , T_2 and T_3 are relation symbols of arities 1, 1, 2, 3 and 4, respectively, and C and D are constant symbols. Let \mathcal{P} be a σ_b -structure. We can think of the elements of $|\mathcal{P}|$ as being the places of a Petri net and the relations P and Q as describing two partitions of these places. We can think of:

- the relation T_1 as describing the set of transitions

$$\{(\{u\}, \{v\}) : u, v \in P \text{ and } T_1(u, v) \text{ holds}\};$$

- the relation T_2 as describing the set of transitions

$$\begin{aligned} & \{(\{u, i\}, \{v, i\}) : u, v \in P, i \notin P, i \in Q \text{ and } T_2(u, v, i) \text{ holds}\} \\ & \cup \{(\{u, j\}, \{v, j\}) : u, v \in P, j \notin P, Q \text{ and } T_2(u, v, j) \text{ holds}\}; \end{aligned}$$

and

- the relation T_3 as describing the set of transitions

$$\{(\{u, i\}, \{v, j\}) : u, v \in P, i, j \notin P, i \in Q, j \notin Q \text{ and } T_3(u, v, i, j) \text{ holds}\}.$$

Furthermore, the initial marking of our Petri consists of the place C and the places not in P but in Q . We define the problem Ω_b as

$\{\mathcal{P} \in \text{STRUCT}(\sigma_b) \ : \ \text{there is a marking reachable from the initial marking in which there is at least one token on the place } D\}$. \square

Note that the transitions encoded within a σ_b -structure \mathcal{P} are of one of four types, as depicted in Fig. 1, and that the relations T_1 , T_2 and T_3 of \mathcal{P} might have additional tuples in them that do not affect how we think of \mathcal{P} as a Petri net.

With reference to our decision problem presented earlier, it should be clear that: the places in P correspond to the vertices V of our digraph $G = (V, E)$, with C corresponding to the source vertex and D the sink vertex; the places not in P but in Q correspond to the user resources; and the places not in P and not in Q correspond to the system resources (henceforth, we shall use this terminology to describe the places of our Petri net). Additionally, the transitions described by T_1 correspond to edges of E with no labels; the transitions described by T_2 yield edges labelled with labels of the form ‘user resource r_i is unused’ and ‘system resource s_j is available’; and the transitions described by T_3 yield edges labelled with labels of the form ‘user resource r_i is unused and this move uses this resource but makes the system resource s_j available’. We interpret a user resource as being in the state ‘unused’, if there is a token on it, and as being in the state ‘used’ otherwise (such places only ever have at most one token on them). It may be the case, in a reachable marking, that a system resource has more than one token on it. However, tokens can not be removed from such places. Thus, it is only ever important as to whether a system resource has no tokens on it, when we think of it being in the state ‘unavailable’, or at least one token on it when, we think of it being in the state ‘available’.

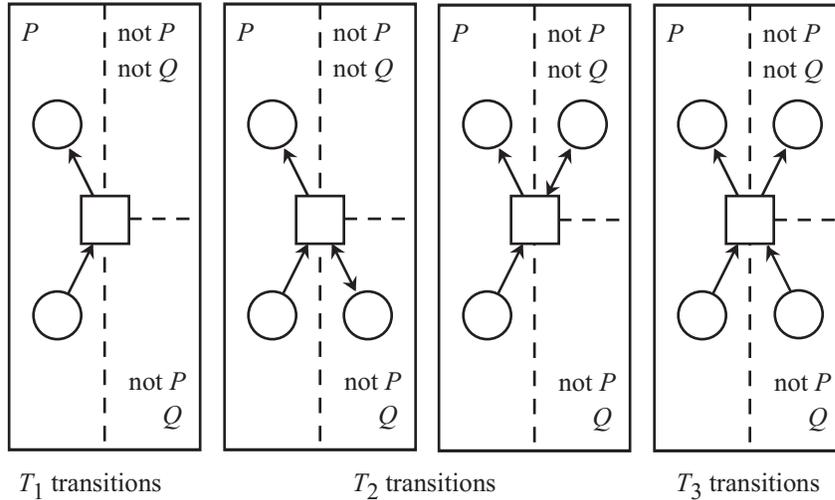


Figure 1. The different types of transitions.

The proof of the following theorem is similar to those in [39] although there are additional complications caused by only having assignments which set array values to *max*.

Theorem 7 *There is a quantifier-free first-order translation with 2 constants from any problem in NPSB(1) to the problem Ω_b . Hence, Ω_b is complete for NPSB(1) via quantifier-free first-order translations with 2 constants.*

Proof Let ρ be a program scheme of NPSB(1) over some signature σ in which if and if-then-else instructions might occur. W.l.o.g., we may assume that array symbols only appear in assignment instructions, that there is only one array symbol, B , and that this array symbol has dimension $d \geq 1$. We assume that the variables involved in ρ are x_1, x_2, \dots, x_k .

Let \mathcal{A} be a σ -structure of size $n \geq 2$. An element $\mathbf{u} = (u_0, u_1, \dots, u_k)$ of $\{1, 2, \dots, l\} \times |\mathcal{A}|^k$ encodes a partial ID of ρ on input \mathcal{A} via: a computation of ρ on \mathcal{A} is about to execute instruction u_0 and the variables x_1, x_2, \dots, x_k currently have the values u_1, u_2, \dots, u_k , respectively. Henceforth, we identify partial IDs of ρ and the elements of $\{1, 2, \dots, l\} \times |\mathcal{A}|^k$.

We now build a Petri net \mathcal{P} , as in Definition 6, using ρ and \mathcal{A} . Our Petri net \mathcal{P} has a set of places consisting of the set $\{1, 2, \dots, l\} \times |\mathcal{A}|^k$ in union with the set $\{\mathbf{w}_0, \mathbf{w}_m : \mathbf{w} \in |\mathcal{A}|^d\}$. The sets of places P and Q are

$$P = \{1, 2, \dots, l\} \times |\mathcal{A}|^k \text{ and } Q = \{\mathbf{w}_0 : \mathbf{w} \in |\mathcal{A}|^d\},$$

respectively. Hence, the user resources are $\{\mathbf{w}_0 : \mathbf{w} \in |\mathcal{A}|^d\}$ and the system resources $\{\mathbf{w}_m : \mathbf{w} \in |\mathcal{A}|^d\}$. We shall use a token on the user resource $(w_1, w_2, \dots, w_d)_0$ to signify that the current value of $B[w_1, w_2, \dots, w_d]$ is 0; and a token on the system resource $(w_1, w_2, \dots, w_d)_m$ to signify that the current value of $B[w_1, w_2, \dots, w_d]$ is *max*. Obviously, we have to take care to ensure that a marking does not yield contradictory interpretations.

Let $\mathbf{u} \in \{1, 2, \dots, l\} \times |\mathcal{A}|^k$.

Suppose that the instruction u_0 does not involve the array symbol B and it is possible for ρ on input \mathcal{A} to move from any ID whose partial ID is \mathbf{u} to an ID whose partial ID is \mathbf{v} in one step. Then the transition $(\{\mathbf{u}\}, \{\mathbf{v}\})$ is in T_1 (more precisely, the pair (\mathbf{u}, \mathbf{v}) is in T_1).

Suppose that the instruction u_0 is of the form $x_j := B[x_{i_1}, x_{i_2}, \dots, x_{i_d}]$ and it is possible for ρ on input \mathcal{A} to move from any ID whose partial ID is \mathbf{u} to an ID whose partial ID is \mathbf{v} in one step (because the value of $B[x_{i_1}, x_{i_2}, \dots, x_{i_d}]$ is such that ρ on input \mathcal{A} can move from an ID whose partial ID is \mathbf{u} to an ID whose partial ID is \mathbf{v} in one step). Then both of the transitions $(\{\mathbf{u}, (u_{i_1}, u_{i_2}, \dots, u_{i_d})_0\}, \{\mathbf{v}, (u_{i_1}, u_{i_2}, \dots, u_{i_d})_0\})$ and $(\{\mathbf{u}, (u_{i_1}, u_{i_2}, \dots, u_{i_d})_m\}, \{\mathbf{v}, (u_{i_1}, u_{i_2}, \dots, u_{i_d})_m\})$ are in T_2 (of course, in the former transition, v_j is 0, and in the latter v_j is *max*, with $u_i = v_i$, for all $i = 1, 2, \dots, k$ different from j).

Suppose that the instruction u_0 is of the form $B[x_{i_1}, x_{i_2}, \dots, x_{i_d}] := \text{max}$ and it is possible for ρ on input \mathcal{A} to move from an ID whose partial ID is \mathbf{u} to an ID whose partial ID is \mathbf{v} in one step. Then the transition $(\{\mathbf{u}, (u_{i_1}, u_{i_2}, \dots, u_{i_d})_m\}, \{\mathbf{v}, (u_{i_1}, u_{i_2}, \dots, u_{i_d})_m\})$ is in T_2 and the transition $(\{\mathbf{u}, (u_{i_1}, u_{i_2}, \dots, u_{i_d})_0\}, \{\mathbf{v}, (u_{i_1}, u_{i_2}, \dots, u_{i_d})_m\})$ is in T_3 (of course, in both transitions $u_i = v_i$, for $i = 1, 2, \dots, k$).

Our initial marking of \mathcal{P} is such that there is one token on each place of $\{\mathbf{w}_0 : \mathbf{w} \in |\mathcal{A}|^d\}$ and one token on the place $(1, \mathbf{0}) \in \{1, 2, \dots, l\} \times |\mathcal{A}|^k$, which we define to be C ; and we define D as the place $(l, \mathbf{max}) \in \{1, 2, \dots, l\} \times |\mathcal{A}|^k$.

It is not difficult to see that our Petri net \mathcal{P} (that is, our σ_b -structure \mathcal{P}) can be described in terms of the σ -structure \mathcal{A} using quantifier-free first-order formulae (in

which 0 and max appear: explicit descriptions of structures by quantifier-free first-order formulae are given in, for example, [34]). Consequently, in order for the result to follow we need to show that: $\mathcal{A} \models \rho$ if, and only if, $\mathcal{P} \in \Omega_b$; and that $\Omega_b \in \text{NPSB}(1)$.

Suppose that $\mathcal{A} \models \rho$. Then there is a sequence π of (full, not partial) IDs starting at the initial ID (where all variables have the value 0, where the instruction to be executed is instruction 1 and where the array B has the value 0 throughout) and ending in a final ID (where all variables have the value max and where the instruction to be executed is instruction l) such that ρ moves from one ID in π to the next in one step. As hinted earlier, we can mirror any ID with a set of markings of our Petri net \mathcal{P} as follows. If the ID consists of the partial ID $\mathbf{u} \in \{1, 2, \dots, l\} \times |\mathcal{A}|^k$ together with some valuation on the array B then the place \mathbf{u} is marked with one token as are the places of $\{\mathbf{w}_0 : \mathbf{w} \in |\mathcal{A}|^d, B[\mathbf{w}] = 0\}$, and the places of $\{\mathbf{w}_m : \mathbf{w} \in |\mathcal{A}|^d, B[\mathbf{w}] = max\}$ are marked with at least one token. This accounts for all tokens. Note that the initial ID of ρ corresponds to the initial marking of \mathcal{P} . A simple analysis yields that if ρ on input \mathcal{A} moves from one ID to another in one step then the Petri net can fire a transition to move from the marking corresponding to the first ID to a marking corresponding to the subsequent ID; and conversely (as remarked earlier, as regards the system resources, it does not matter how many tokens reside on them but only whether or not at least one token resides). Hence, $\mathcal{A} \models \rho$ if, and only if, $\mathcal{P} \in \Omega_b$.

All that remains is to show that $\Omega_b \in \text{NPSB}(1)$. There are two essential difficulties in deriving a program scheme to accept Ω_b . First, a σ_b -structure \mathcal{P} might be such that a reachable marking involves more than one token on some system resource; and we need to cater for this event when we simulate a sequence of transitions in \mathcal{P} by an execution of a program scheme on input \mathcal{P} . Second, we need to keep track of where tokens are in a way which avoids us modelling the fact that a token is on a place simply by using an array indexed by the place names; for we are not allowed to register that a token has moved from a place by assigning some array element the value 0 (recall, the only assignment instruction allowed on an array element is to set that element to max).

Our Petri net \mathcal{P} is such that initially there is one token, call it t , on the place C of P and there is one token on every user resource (we assume that the place C is indeed in P : otherwise, our program scheme simply rejects the input \mathcal{P}). No other tokens are involved in the initial marking. Also, transitions are such that we can imagine the token t as being moved from place to place amongst the places of P , and we can imagine every other token either staying where it is, after some transition, or being moved from user resource to a system resource, and then staying where it is thereafter.

As regards our first difficulty, we do not need to actually monitor how many tokens lie on any system resource but only whether there is at least one token such a place. This obviates the need to count tokens. As regards our second difficulty, in order to decide whether (at least) one token lies on some system resource s , we use a dedicated array B_1 , of dimension 1, so that whenever a token is placed on such a s then $B_1[s]$ is set at max : once $B_1[s]$ has been set to max we know that there will be a token on s thereafter. In order to decide whether a token lies on some user resource r , we use an array B_2 , of dimension 1, to register when the token originally on the place r is first moved from r by setting $B_2[r]$ equal to max at this point. Consequently, if we wish to know whether there is a token on such a place r , we test to see whether

$B_2[r] = 0$ holds. Finally, we model the movement of the solitary token t by using a dedicated variable, x say: that is, the token t is on place p if, and only if, x has the value p . Given the above discussion, it is straightforward to see that the problem Ω_b can be accepted by a program scheme of NPSB(1), and so the result follows. \square

In essence, Theorem 7 tells us that any problem accepted by a program scheme of NPSB(1) can be described by a sentence of the form

$$\Omega_b[\lambda \mathbf{x} \psi_P(\mathbf{x}), \mathbf{x} \psi_Q(\mathbf{x}), \mathbf{x}, \mathbf{y} \psi_1(\mathbf{x}, \mathbf{y}), \mathbf{x}, \mathbf{y}, \mathbf{z} \psi_2(\mathbf{x}, \mathbf{y}, \mathbf{z}), \\ \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{w} \psi_3(\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{w})](\mathbf{u}, \mathbf{v}),$$

where: $|\mathbf{x}| = |\mathbf{y}| = |\mathbf{z}| = |\mathbf{w}| = k$, for some $k \geq 1$, and all variables are distinct; ψ_P , ψ_Q , ψ_1 , ψ_2 and ψ_3 are quantifier-free first-order formulae over $\sigma_b \cup \{0, max\}$; and \mathbf{u} and \mathbf{v} are k -tuples of constant symbols (in fact, we can actually take \mathbf{u} to be 0 repeated k times and \mathbf{v} to be max repeated k times: moreover, the sentence is such that whether it is true in some given structure is independent of the distinct values chosen for 0 and max).

Similarly to as in [39], Theorem 7 allows us to relate the class of problems accepted by the program schemes of NPSB with the class of problems defined by the sentences of the logic $(\pm\Omega_b)^*[\text{FO}]$. For each $m \geq 1$, we define the fragment $\pm\Omega_b(m)$ of $(\pm\Omega_b)^*[\text{FO}]$ as follows.

- $\pm\Omega_b(1)$ consists of all formulae of the form

$$\Omega_b[\lambda \mathbf{x} \psi_P, \mathbf{x} \psi_Q, \mathbf{x}, \mathbf{y} \psi_1, \mathbf{x}, \mathbf{y}, \mathbf{z} \psi_2, \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{w} \psi_3](\mathbf{u}, \mathbf{v}),$$

where: ψ_P , ψ_Q , ψ_1 , ψ_2 and ψ_3 are quantifier-free first-order formulae over $\sigma_b \cup \{0, max\}$; \mathbf{u} and \mathbf{v} are k -tuples of constant symbols or variables; there may be other free variables; and the truth of any interpretation of the formula (over a relevant structure and with values given for any free variables) is independent of the pair of distinct values chosen for 0 and max .

- $\pm\Omega_b(m+1)$, for odd $m \geq 1$, consists of the universal closure of $\pm\Omega_b(m)$; that is, the set of formulae of the form $\forall z_1 \forall z_2 \dots \forall z_k \psi$, where ψ is a formula of $\pm\Omega_b(m)$.
- $\pm\Omega_b(m+1)$, for even $m \geq 2$, consists of the set of formulae of the form

$$\Omega_b[\lambda \mathbf{x}(\psi_P^1 \vee \neg \psi_P^2), \mathbf{x}(\psi_Q^1 \vee \neg \psi_Q^2), \mathbf{x}, \mathbf{y}(\psi_1^1 \vee \neg \psi_1^2), \mathbf{x}, \mathbf{y}, \mathbf{z}(\psi_2^1 \vee \neg \psi_2^2), \\ \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{w}(\psi_3^1 \vee \neg \psi_3^2)](\mathbf{u}, \mathbf{v}),$$

where: ψ_P^1 , ψ_P^2 , ψ_Q^1 , ψ_Q^2 , ψ_1^1 , ψ_1^2 , ψ_2^1 , ψ_2^2 , ψ_3^1 and ψ_3^2 are formulae of $\pm\Omega_b(m)$; \mathbf{u} and \mathbf{v} are tuples of constant symbols or variables; there may be other free variables; and the truth of any interpretation of the formula (over a relevant structure and with values given for any free variables) is independent of the pair of distinct values chosen for 0 and max .

As in [36], a straightforward induction yields that:

- for every odd $m \geq 1$, every formula in the closure of $\pm\Omega_b(m)$ under \wedge , \vee and \exists is logically equivalent to a formula of $\pm\Omega_b(m)$; and

- for every even $m \geq 1$, every formula in the closure of $\pm\Omega_b(m)$ under \wedge , \vee and \forall is logically equivalent to a formula of $\pm\Omega_b(m)$.

Consequently, $(\pm\Omega_b)^*[\text{FO}] = \cup\{\Omega_b(m) : m \geq 1\}$.

Corollary 8 *In the presence of 2 built-in constant symbols, $\pm\Omega_b(m) = \text{NPSB}(m)$, for each $m \geq 1$; and so $(\pm\Omega_b)^*[\text{FO}] = \text{NPSB}$.*

Proof We proceed by induction on m almost identically to the proof of Corollary 10 from [39]. The base case, when $m = 1$, follows by Theorem 7. \square

Note that $(\pm\Omega_b)^*[\text{FO}] = \text{NPSB}$ even in the absence of our 2 built-in constant symbols as we can ‘build them ourselves’ using existential quantification.

We end this section by showing that NPSB can not be realized as a fragment of $\mathcal{L}_{\infty\omega}^\omega$ (unlike NPS and NPSS).

Lemma 9 *The problem CUB can be accepted by a program scheme of $\text{NPSB}(1)$.*

Proof It was shown in [39] that CUB is in $\text{NPSA}(1)$: however, the program scheme used there to accept CUB is not in $\text{NPSB}(1)$. Nevertheless, the basic approach can be amended to yield a program scheme of $\text{NPSB}(1)$.

Let \mathcal{G} be a σ_2 -structure. We begin by ‘guessing’ a set of distinct edges in the graph \mathcal{G} . We use two 3-dimensional array symbols, B_1 and B_2 , to store these guessed edges. In particular, if our first guessed edge is (u_1, v_1) , having checked that (u_1, v_1) is indeed an edge of \mathcal{G} , we set $B_1[0, 0, u_1] = \text{max}$ and $B_2[0, 0, v_1] = \text{max}$. Next, we guess an edge (u_2, v_2) , check to see whether this edge is indeed an edge of \mathcal{G} and then check to see whether this edge is different from (u_1, v_1) . If so then we set $B_1[u_1, v_1, u_2] = \text{max}$ and $B_2[u_1, v_1, v_2] = \text{max}$: otherwise, we set $B_1[u_1, v_1, \text{max}] = \text{max}$ and $B_2[u_1, v_1, \text{max}] = \text{max}$ and stop guessing. We continue in this fashion until the guessing stage stops whence we have a list of distinct edges of \mathcal{G} .

Finally, we check to see whether the guessed set of edges induces a regular subgraph of \mathcal{G} of degree 3. It is clear that this whole process can be implemented by a program scheme of $\text{NPSB}(1)$: hence, the result follows. \square

The facts that the problem CUB can not be defined in $\mathcal{L}_{\infty\omega}^\omega$ (see [37]) and that non-recursive problems can be defined in $\mathcal{L}_{\infty\omega}^\omega$ (see [12]) immediately yield the following result.

Corollary 10 *There are problems definable in $\text{NPSB}(1)$ (and so NPSB) which are not definable in $\mathcal{L}_{\infty\omega}^\omega$; and there are problems definable in $\mathcal{L}_{\infty\omega}^\omega$ which are not definable in NPSB .* \square

Let us remark that Lemma 9 is subsumed by a later result; nevertheless, we include it here in order to show that $\text{NPSB}(1)$ can not be regarded as a fragment of $\mathcal{L}_{\infty\omega}^\omega$ (which is our concern at the moment).

5 Ordered structures

Given our characterization of the class of problems accepted by the program schemes of NPSB, we now consider the class of problems accepted by these program schemes when we restrict ourselves to ordered structures.

Using Theorem 1, we can easily modify the program scheme implicit in the proof of Lemma 9 so that, in the presence of a built-in successor relation, it accepts any given problem in **NP**. Conversely, any problem in $\text{NPSB}_s(1)$ is in **NP**. Theorem 7 then yields the following result.

Corollary 11 *As classes of problems, $\text{NP} = \text{NPSB}_s(1)$; and Ω_b is complete for **NP** via quantifier-free first-order translations with successor.* \square

By Corollary 8, $\text{NPSB}_s = (\pm\Omega_b)^*[\text{FO}_s]$; and by Corollary 5.5 of [32] and Corollary 11, $(\pm\Omega_b)^*[\text{FO}_s] = (\pm\text{HP})^*[\text{FO}_s]$, where HP is the problem over the signature σ_{2++} consisting of all those σ_{2++} -structures \mathcal{A} which, when considered as digraphs with edge relation $E^{\mathcal{A}}$ and two given vertices $C^{\mathcal{A}}$ and $D^{\mathcal{A}}$, are such that there is a Hamiltonian path from $C^{\mathcal{A}}$ to $D^{\mathcal{A}}$. Furthermore, by Corollary 3.2.2 of [35], $(\pm\text{HP})^*[\text{FO}_s] = \mathbf{L}^{\text{NP}}$ (the class of problems accepted by a log-space deterministic oracle Turing machine with access to an **NP** oracle), and every problem in $(\pm\text{HP})^*[\text{FO}_s]$ can be defined by a sentence of the form:

$$\exists z_1 \exists z_2 \dots \exists z_m (\text{HP}[\lambda \mathbf{x}, \mathbf{y} \psi(\mathbf{x}, \mathbf{y}, \mathbf{z})](\mathbf{0}, \mathbf{max}) \wedge \neg \text{HP}[\lambda \mathbf{x}, \mathbf{y} \varphi(\mathbf{x}, \mathbf{y}, \mathbf{z})](\mathbf{0}, \mathbf{max})),$$

where: \mathbf{x} and \mathbf{y} are k -tuples of variables, for some k ; ψ and φ are quantifier-free first-order formulae (with successor); and $\mathbf{0}$ (resp. \mathbf{max}) is the constant symbol 0 (resp. max) repeated k times. Hence, translating this normal form into a program scheme yields that any problem in NPSB_s can actually be accepted by a program scheme of $\text{NPSB}_s(3)$. Furthermore, any problem accepted by a program scheme $\forall z_1 \forall z_2 \dots \forall z_m \rho$ of $\text{NPSB}_s(2)$ can be accepted by a program scheme of $\text{NPSB}_s(1)$: we simply replace the universal quantification by code within a program scheme of $\text{NPSB}_s(1)$ which uses a while instruction and the successor relation to check whether a structure is accepted by ρ for every valuation of the free variables z_1, z_2, \dots, z_m . Hence, we have the following result.

Theorem 12 $\text{NPSB}_s(1) = \text{NPSB}_s(2) = \text{NP}$ and $\text{NPSB}_s(3) = \text{NPSB}_s = \mathbf{L}^{\text{NP}}$. \square

We now turn to the relative computational capabilities of the classes of program schemes $\text{NPSB}(1)$ and $\text{NPSA}(1)$ on the class of all finite structures (we have more to say about comparing the classes NPSB and NPSA in the Conclusion).

The following definitions are essential to what follows. Let σ be some signature and let \mathcal{A} and \mathcal{B} be σ -structures. If $|\mathcal{A}| \subseteq |\mathcal{B}|$ and:

- for every relation symbol R of σ , $R^{\mathcal{A}}$ is $R^{\mathcal{B}}$ restricted to $|\mathcal{A}|$; and
- for every constant symbol C of σ , $C^{\mathcal{A}} = C^{\mathcal{B}}$,

then we say that \mathcal{A} is a *sub-structure* of \mathcal{B} and write $\mathcal{A} \subseteq \mathcal{B}$. If the problem Ω over σ is such that for all σ -structures \mathcal{A} and \mathcal{B} for which $\mathcal{A} \subseteq \mathcal{B}$, it is necessarily the case that $\mathcal{A} \in \Omega$ implies $\mathcal{B} \in \Omega$, then we say that Ω is *closed under extensions*. Let $\mathcal{EX}\mathcal{T}$ be the class of all problems that are closed under extensions.

Lemma 13 *Every problem in $NPSA(1)$ is closed under extensions.*

Proof Let Ω be a problem over the signature σ accepted by the program scheme ρ of $NPSA(1)$. Let \mathcal{A} and \mathcal{B} be σ -structures such that $\mathcal{A} \subseteq \mathcal{B}$, and suppose that $\mathcal{A} \models \rho$. Consider the program scheme ρ on input \mathcal{B} where 0 and max are chosen to be distinct elements of $|\mathcal{A}|$. By ‘mirroring’ an accepting computation of ρ on input \mathcal{A} , with the chosen 0 and max , we obtain an accepting computation of ρ on input \mathcal{B} (the fact that all tests in while, if and if-then-else instructions are quantifier-free first-order enables us to do this). Hence, $\mathcal{B} \in \Omega$. \square

Theorem 14 $PSPACE \cap \mathcal{EXPT} = NPSA(1)$ and $NP \cap \mathcal{EXPT} = NPSB(1)$.

Proof Let Ω be some problem in $PSPACE \cap \mathcal{EXPT}$. By [33], there exists a program scheme $\rho \in NPSA_s(1)$ accepting Ω . Modify ρ to obtain the program scheme $\rho' \in NPSA(1)$ as follows. In ρ' , begin by guessing a successor relation; that is, when \mathcal{A} is some input structure, guess elements $u_1, u_2, \dots, u_m \in |\mathcal{A}|$ so that

$$M[0] = u_1, M[u_1] = u_2, \dots, M[u_m] = max,$$

where M is a new one-dimensional array symbol and where the elements of $\{0, u_1, u_2, \dots, u_m, max\}$ are distinct (this latter condition can be checked as we guess). Replace any atomic relation of the form $succ(x, y)$ in ρ with the formula $y = M[x]$, and replace any instruction of the form **guess** x with the following fragment of code:

```

guess  $x$ 
 $goodx := 0$ 
 $ok := 0$ 
while  $ok = 0$  do
  if  $(x = goodx \vee goodx = max)$  then
     $ok := max$ 
  else
     $goodx := M[goodx]$ 
  fi
od
if  $x \neq goodx$  then ‘loop forever’ fi

```

(where $goodx$ and ok are new variables). Note that this fragment of code essentially limits our guesses to elements appearing in the domain of our guessed successor relation. We need to show that acceptance by the program scheme ρ' is invariant with respect to 0 and max and that it accepts the problem Ω .

Suppose that $\mathcal{A} \in \Omega$. Then \mathcal{A} is accepted by ρ no matter which successor relation is chosen for $succ$ in ρ . Choose distinct $0'$ and max' in $|\mathcal{A}|$ and a successor relation $succ'$ on $|\mathcal{A}|$ (with minimal and maximal elements the chosen elements $0'$ and max'). In particular, ρ accepts \mathcal{A} with these constants and this successor relation. Consider a computation of ρ' on input \mathcal{A} where the guessed successor relation is $succ'$. Then there exists a computation of ρ' mirroring any accepting computation of ρ on input \mathcal{A} with this particular successor relation. That is, \mathcal{A} is accepted by ρ' and acceptance does not depend upon the chosen constants 0 and max .

Conversely, suppose that there is a guessed successor relation, call it $succ'$ (whose domain need not be all of $|\mathcal{A}|$), with minimal and maximal elements $0'$ and max' ,

yielding an accepting computation of ρ' on input \mathcal{A} . Let $B \subseteq |\mathcal{A}|$ be the domain of this successor relation and let \mathcal{B} be the restriction of \mathcal{A} to B . Then \mathcal{B} is accepted by ρ when the successor relation is taken as succ' (note that the domain of succ' is the whole of $|\mathcal{B}|$). Hence, $\mathcal{B} \in \Omega$. However, Ω is closed under extensions and so $\mathcal{A} \in \Omega$. But we have seen from above that if $\mathcal{A} \in \Omega$ then \mathcal{A} is accepted by ρ' and acceptance does not depend upon the chosen constants 0 and max . Thus, acceptance by ρ' is invariant with respect to 0 and max ; and $\mathbf{PSPACE} \cap \mathcal{EXPT} \subseteq \text{NPSA}(1)$. The fact that every problem in $\text{NPSA}(1)$ can be solved by a polynomial-space algorithm is straight-forward; and every problem in $\text{NPSA}(1)$ is closed under extensions by Lemma 13.

Now consider a problem $\Omega \in \mathbf{NP} \cap \mathcal{EXPT}$ accepted by the program scheme $\rho \in \text{NPSB}_s(1)$. We proceed as above, and define a program scheme $\rho' \in \text{NPSB}(1)$, except with the following amendment. In $\text{NPSB}(1)$, we are only allowed assignments to array elements of the form $M[x_1, x_2, \dots, x_k] := \text{max}$ and so we need some way of encoding our guessed successor relation. We encode our relation as:

$$M[0, u_1] = \text{max}, M[u_1, u_2] = \text{max}, \dots, M[u_m, \text{max}] = \text{max},$$

where M is a new array symbol of dimension 2. Of course, we ensure that the elements of $\{0, u_1, u_2, \dots, u_m, \text{max}\}$ are distinct as we guess. Note that we need to remember the previously guessed element, u_i , so that we know to set $M[u_i, u_{i+1}]$ equal to max . We also need to modify our code so that an atomic relation of the form $\text{succ}(x, y)$ is replaced by the formula $M[x, y] = \text{max}$. Arguing as above yields the result. \square

Note that the construction used in the proofs of Lemma 9 and Theorem 14 (right at the end) can easily be generalized to show that the ‘logic’ formed by allowing any program scheme ρ of $\text{NPSB}(1)$ to be prefixed with a sequence of existentially-quantified relation symbols (where these relation symbols do not appear in the underlying signature of the problem in hand but do appear in the program scheme ρ) captures the class of problems $\text{NPSB}(1)$. To see this, we simply ‘guess’ relations for these quantified relation symbols and ‘string’ each relation together using an appropriate array in the style of a linked list. Should we need to check whether some tuple of values is in the relation in some computation of the program scheme then we work down the linked list looking for this tuple. Consequently, we have the following result, which gives some idea as to the power of $\text{NPSB}(1)$.

Corollary 15 *$\text{NPSB}(1)$ is closed under existential quantification.*

One view of Theorem 14 is that it provides syntactic characterizations (via the the classes of program schemes $\text{NPSA}(1)$ and $\text{NPSB}(1)$) of semantically defined complexity classes (namely, $\mathbf{PSPACE} \cap \mathcal{EXPT}$ and $\mathbf{NP} \cap \mathcal{EXPT}$). Actually, we make this remark modulo our earlier discussion where we mention that we can indeed obtain a true ‘syntactic’ characterization by working with variables x_0 and x_{max} rather than using built-in constant symbols 0 and max .

Corollary 16 $\mathbf{NP} = \mathbf{PSPACE}$ if, and only if, $\text{NPSA}(1) = \text{NPSB}(1)$.

Proof If $\mathbf{NP} = \mathbf{PSPACE}$ then $\mathbf{NP} \cap \mathcal{EXPT} = \mathbf{PSPACE} \cap \mathcal{EXPT}$; and so $\text{NPSA}(1) = \text{NPSB}(1)$ by Theorem 14. Conversely, if $\text{NPSA}(1) = \text{NPSB}(1)$ then $\text{NPSA}_s(1) = \text{NPSB}_s(1)$; and so $\mathbf{NP} = \mathbf{PSPACE}$ by [33] and Corollary 11. \square

Corollary 16 relates an open complexity-theoretic question with a question involving two classes of problems, each problem of which has asymptotic probability 1. This result could easily have been obtained simply by observing that there are **PSPACE**-complete that are closed under extensions. Nevertheless, we include it here to emphasise the relationship between standard complexity classes and our classes of program schemes.

Corollary 16 can be extended slightly in that we can obtain some additional equivalences involving fragments of certain vectorized Lindström logics. Referring back to Theorem 5, the problem mentioned in that theorem is actually defined as follows.

Definition 17 Let the signature $\sigma_a = \langle T_1, T_3, M, C \rangle$, where M is a unary relation symbol, T_1 is a binary relation symbol, T_3 is a relation symbol of arity 4 and C is a constant symbol. We can envisage a σ_a -structure \mathcal{A} as a Petri net whose places are given by $|\mathcal{A}|$ and whose transitions are given by T_1 and T_2 via:

- there is a transition $(\{x\}, \{y\})$ whose input place is $\{x\}$ and whose output place is $\{y\}$ if, and only if, $T_1(x, y)$ holds; and
- there is a transition $(\{x_1, x_2\}, \{y_1, y_2\})$ whose input places are $\{x_1, x_2\}$ and whose output places are $\{y_1, y_2\}$ if, and only if, $T_3(x_1, x_2, y_1, y_2)$ holds, where $x_1 \neq x_2$ and $y_1 \neq y_2$.

The relation M can be seen as providing an initial marking (with one token on place p if, and only if, $M(p)$ holds) and the constant C as providing a final marking (consisting of one token on the place C).

A σ_a -structure \mathcal{A} , *i.e.*, a Petri net, complete with initial and final markings, where every transition has either 2 input places and 2 output places or 1 input place and 1 output place, is in the problem Ω_a if, and only if, there is a marking covering the final marking that is reachable from the initial marking, *i.e.*, there is a reachable marking in which there is at least one token on the place C . \square

Corollary 18 *The following are equivalent.*

- (a) **NP = PSPACE**.
- (b) $NPSB(1) = NPSA(1)$.
- (c) $\Omega_b^1[FO] = \Omega_a^1[FO]$.
- (d) *The problems Ω_b and Ω_a are equivalent via quantifier-free first-order translations with 2 constants.*

Proof Corollary 16 implies (a) \Leftrightarrow (b). Theorems 5 and 7 imply (b) \Leftrightarrow (d). It is trivially the case that (c) \Rightarrow (a) and that (d) \Rightarrow (c). \square

We end by returning to an earlier remark concerning the NPSB hierarchy on the class of all finite structures. We include the following result here as we can utilize results of this section, and this result also applies to the NPSA hierarchy.

Proposition 19 *On the class of all finite structures,*

$$NPSB(1) \subset NPSB(2) \subset NPSB(3).$$

Proof By Lemma 13, every problem in NPSB(1) is closed under extensions; and so, trivially, $\text{NPSB}(1) \subset \text{NPSB}(2)$.

Consider the following first-order sentence over the signature $\sigma_2 = \langle E, C \rangle$, where E is a binary relation symbol and C is a constant symbol:

$$\exists x(E(C, x) \wedge \exists y(E(x, y) \wedge \forall z(E(x, z) \Rightarrow z = y))).$$

There is clearly a program scheme of NPSB(3) accepting the problem Ω defined by this sentence. For any $k \geq 1$, consider the digraphs, \mathcal{A}_k and \mathcal{B}_k , depicted in Fig. 2 (note that \mathcal{B}_k only differs from \mathcal{A}_k by having an extra vertex and edge). No matter what the value of k , $\mathcal{A}_k \in \Omega$ but $\mathcal{B}_k \notin \Omega$. We shall show that for any program scheme ρ of NPSB(2), there exists some k such that $\mathcal{A}_k \models \rho$ implies that $\mathcal{B}_k \models \rho$. This will yield our result.

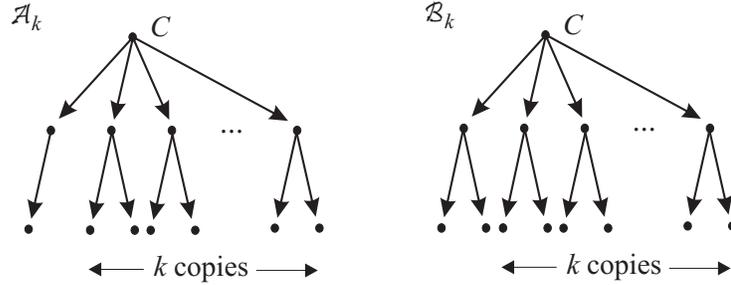


Figure 2. The digraphs \mathcal{A}_k and \mathcal{B}_k .

Let ρ be a program scheme of NPSB(2) of the form $\forall x_k \forall x_2 \dots \forall x_k \rho'$, for some program scheme ρ' of NPSB(1), and let $\tilde{\mathcal{B}}_k$ be an extension of \mathcal{B}_k by k constants (one for each variable x_i). There is an extension of \mathcal{A}_k , denote it $\tilde{\mathcal{A}}_k$, such that $\tilde{\mathcal{A}}_k$ is embeddable into $\tilde{\mathcal{B}}_k$ via a one-to-one mapping; call the mapping π . Suppose that there is an accepting computation of ρ' on input $\tilde{\mathcal{A}}_k$. We can ‘mirror’ this computation by a computation of ρ' on $\tilde{\mathcal{B}}_k$ by making guesses according to the mapping π (after having chosen our constants 0 and *max*, again according to π). The two computations of ρ' , on $\tilde{\mathcal{A}}_k$ and $\tilde{\mathcal{B}}_k$, proceed in tandem (in that their flows of control are identical) and because the computation of ρ' on $\tilde{\mathcal{A}}_k$ leads to acceptance, so must the computation of ρ' on $\tilde{\mathcal{B}}_k$ (recall, any tests are quantifier-free first-order and so only ever refer to the current values of variables). Our result follows. \square

We add that the proof of Proposition 19 suffices to show that, on the class of all finite structures, $\text{NPSA}(1) \subset \text{NPSA}(2) \subset \text{NPSA}(3)$.

6 Amended semantics

Finally, let us amend our semantics of the program schemes of NPSB. When we defined the semantics of a program scheme ρ of NPSB($2i + 1$), for some $i > 0$, we insisted that when a test in some if-then-else or while instruction is evaluated (recall, such a test is a program scheme of NPSB($2i$)), the only values used in this evaluation are the current values of the variables of ρ . In particular, all arrays involved in the evaluation

are initialized to 0 prior to the evaluation. Suppose that we now insist that arrays used in the evaluation are *initialized to their current values* prior to the evaluation. Consequently, not only can we pass the current values of the variables across to an evaluation, we can pass the current values of the arrays across too (or course, when the program scheme ρ resumes after evaluation of the test, the values of the arrays are what they were prior to the evaluation of the test). We denote the program schemes of NPSB with this semantics as NPSB^p to reflect the fact that a polynomial number of values is passed across in an evaluation (rather than just a constant number in the standard semantics). Allowing a polynomial number of values to be passed across to an evaluation drastically changes the expressibility of the resulting class of program schemes (modulo the usual complexity-theoretic qualifications). The complexity class **PH** is the *Polynomial Hierarchy*; that is, $\mathbf{PH} = \cup_{i=1}^{\infty} \Sigma_i^p$, where $\Sigma_1^p = \mathbf{NP}$ and where, for each $i \geq 2$, $\Sigma_i^p = \mathbf{NP}^{\Sigma_{i-1}^p}$ (the class of problems accepted by a polynomial-time non-deterministic oracle Turing machine with access to a Σ_{i-1}^p oracle).

Theorem 20 $\text{NPSB}^p(1) = \text{NPSB}(1)$, $\text{NPSB}^p(2) = \text{NPSB}(2)$ and for every $i \geq 2$, $\text{NPSB}^p(2i-1) = \text{NPSB}^p(2i) = \Sigma_i^p$. Consequently, $\text{NPSB}^p = \mathbf{PH}$.

Proof Similarly to the proof (elucidated immediately prior to Theorem 12) that $\text{NPSB}_s(1) = \text{NPSB}_s(2)$, so we can show that $\text{NPSB}^p(2i-1) = \text{NPSB}^p(2i)$, for all $i \geq 2$. Obviously, $\text{NPSB}^p(1) = \text{NPSB}(1)$ and $\text{NPSB}^p(2) = \text{NPSB}(2)$ (as our original semantics and our amended semantics do not differ in these cases).

We now show how to build our own successor relation using a program scheme of $\text{NPSB}^p(3)$. Essentially, we guess a successor relation and store it in the array S , of dimension 2, via the following code:

```

x := 0
while x ≠ max do
  guess y
  if x ≠ y then
    S[x, y] := max
    x := y
  fi
od

```

Then we check, using an if-then-else instruction with the test a program scheme of $\text{NPSB}^p(2)$, that every value appears in the guessed relation S and that no value appears more than once. Consequently, by Corollary 11, any problem in **NP** can be accepted by some program scheme of $\text{NPSB}^p(3)$.

Notwithstanding the above remark, we would like to explicitly simulate a non-deterministic polynomial-time Turing machine computation using a program scheme of $\text{NPSB}^p(3)$. We can use arrays to store the work-tape of any such Turing machine and our successor relation, held in S , to mirror the movement of the tape heads. Our only restriction to this simulation is that we can only set array values at *max*: we can not reset them to 0. Hence, the obvious means of simulation is doomed to failure given that, in general, the contents of a cell of a Turing machine work-tape fluctuate and that if we simulate a cell of the work tape using a fixed number of array elements then we can only register a constant number of changes to the cell contents. However,

we can get round this difficulty by using the fact that any (accepting) computation of our Turing machine has length polynomial in the size of the input structure: hence, we can use an array to store the complete history of changes to the contents of a Turing machine work-tape cell as follows.

For simplicity, assume that we wish to hold the contents of n Turing machine work-tape cells (where the input structure has size n) using some arrays and that these contents are only ever 0 or 1. Furthermore, assume that the time taken by our Turing machine to accept (if it does) is n . The general case where a cell can contain more symbols, where there is a polynomial number of work-tape cells to deal with and where the Turing machine accepts in a polynomial number of steps can be handled similarly by increasing the dimensions of our arrays. Let A and B be array symbols of dimension 2. Using our successor relation (constructed earlier), we use the array cells $A[u, 1], A[u, 2], \dots, A[u, n]$ (we think of the elements of our input structure as being named $\{1, 2, \dots, n\}$ with the names reflecting our successor relation) to register the first change of the contents of the work-tape cell u , the second change of the work-tape cell u , the third change of the work-tape cell u , and so on; and the array cells $B[u, 1], B[u, 2], \dots, B[u, n]$ to register the value of work-tape cell u after the first change, the value of work-tape cell u after the second change, the value of work-tape cell u after the third change, and so on.

If $A[u, i] = \text{max}$ then this is interpreted as meaning that there have been at least i changes of contents; and if $B[u, i] = 0$ (resp. $B[u, i] = \text{max}$) then this is interpreted as meaning that after the i th change, the contents of work-tape cell u is 0 (resp. 1). Note that when the work-tape cell u changes from 1 to 0, on the i th change, say, in order to register this change we need only set $A[u, i] = \text{max}$ and leave $B[u, i]$ alone (as it has been initialized to 0). Furthermore, with this representation, and using our successor relation, we can easily determine the current contents of any work-tape cell: we simply cycle down the array A to find the last change of contents and then ascertain the current contents using B . Thus, it should be clear how we can explicitly simulate our Turing machine computation using a program scheme of $\text{NPSB}^p(3)$.

Now, consider a polynomial-time non-deterministic oracle Turing machine M consulting an \mathbf{NP} oracle. By Corollary 11, and using an array to hold the contents of the oracle tape, we can simulate an oracle call of M by an if-then-else instruction where the test is a program scheme of $\text{NPSB}^p(2)$ (exactly because we are allowed, in our modified semantics, to pass the values of arrays over to the evaluation of a test). Hence, we have essentially proven that any problem in $\mathbf{NP}^{\mathbf{NP}}$ can be accepted by a program scheme of $\text{NPSB}^p(3)$. Conversely, it is straightforward to see that any problem accepted by a program scheme of $\text{NPSB}^p(3)$ can be accepted by a polynomial-time non-deterministic oracle Turing machine with an oracle in \mathbf{NP} (the only point worthy of note in this regard is that we must ensure that the contents of all arrays in the program scheme are written on the simulating Turing machine's oracle tape). Hence, $\text{NPSB}^p(3) = \mathbf{NP}^{\mathbf{NP}}$.

The general result now follows by a simple induction: for example, any polynomial-time non-deterministic oracle Turing machine consulting an oracle in $\mathbf{NP}^{\mathbf{NP}}$ can be explicitly simulated; and by above the oracle calls can be simulated by if-then-else instructions where the tests are program schemes from $\text{NPSB}^p(4)$. \square

7 Conclusions

In this paper, we have examined the computational capabilities of different classes of program schemes, based around ‘binary write-once arrays’, on the class of finite structures, the class of ordered finite structures and with respect to different semantics. We now discuss some potential directions for future research.

Perhaps the most obvious unanswered question is as regards the NPSB hierarchy: ‘Is it the case that, like the NPS and NPSS hierarchies, the NPSB hierarchy is proper at every level?’ (the same question can be asked for the NPSA hierarchy). So far, we have not been able to answer this question (beyond Proposition 19). The main reason for the lack of progress is that whereas in [7] we were able to ‘re-use’ domain elements so as to ‘mirror’ computations of program scheme of NPS and NPSS (in the style of the proof of Proposition 19), the existence of arrays means that we can ‘remember the values already used’ in a computation and consequently it is not clear that domain elements can be re-used in a suitably anonymous fashion (the reader is referred to [7], and the proofs therein, in order to make more sense of this remark). The fact that working with program schemes of NPSB takes us outside the ‘bounded-variable world’ of the logic $\mathcal{L}_{\infty\omega}^\omega$ (see Corollary 10), whereas this is not the case with the program schemes of NPS and NPSS, is particularly intriguing in this respect.

In relation to the above comments (and as suggested by an anonymous referee), it would be interesting to further examine the relationship between the classes of program schemes in this paper and (fragments of) the more standard logics from finite model theory and descriptive complexity, such as bounded-variable infinitary logic and second-order logic. For example, how does the class of existential second-order formulae in which the first-order matrix is purely existential compare with the class of program schemes NPSB(1)? Also, can we translate program schemes into restricted infinitary formulae (not necessarily involving a finite number of variables) and apply known results concerning such formulae with certain quantifier alternations to obtain proper hierarchies of program schemes?

The results in Section 6, relating the computational capabilities of the classes of program schemes NPSB(1) and NPSA(1), are in the style of Abiteboul and Vianu [2, 4], Abiteboul, Vianu and Vardi [6] and Dawar [11]. However, we would prefer to have determined similar results but regarding the classes NPSB and NPSA (or, equivalently, the logics $(\pm\Omega_b)^*[\text{FO}]$ and $(\pm\Omega_a)^*[\text{FO}]$). So far, we have been unable to extend the results of Section 6 to these classes of programs schemes. There are some very straightforward implications to be made however. For instance (on the class of all finite structures):

- by Corollary 18, if $\mathbf{NP} = \mathbf{PSPACE}$ then $\text{NPSB} = \text{NPSA}$ (and, equivalently, $(\pm\Omega_b)^*[\text{FO}] = (\pm\Omega_a)^*[\text{FO}]$);
- by Theorems 5 and 12, if $\text{NPSB} = \text{NPSA}$ (or, equivalently, $(\pm\Omega_b)^*[\text{FO}] = (\pm\Omega_a)^*[\text{FO}]$) then $\mathbf{L}^{\mathbf{NP}} = \mathbf{PSPACE}$; and
- by Theorems 5 and 12, if $\Omega_b^*[\text{FO}] = \Omega_a^*[\text{FO}]$ then $\mathbf{NP} = \mathbf{PSPACE}$ (as any problem in $\Omega_b^*[\text{FO}]$ can easily be seen to be in \mathbf{NP}).

We would like to be able to equate the questions: ‘Is $\mathbf{L}^{\mathbf{NP}}$ equal to \mathbf{PSPACE} ?’, ‘Is NPSB equal to NPSA?’ and ‘Is $(\pm\Omega_b)^*[\text{FO}]$ equal to $(\pm\Omega_a)^*[\text{FO}]$?’; as well as the

questions: ‘Is **NP** equal to **PSPACE**?’ and ‘Is $\Omega_b^*[\text{FO}]$ equal to $\Omega_a^*[\text{FO}]$?’. As yet, we have been unable to do so.

Finally, let us return to the decision problem described at the beginning of Section 4 involving the traversal of a digraph subject to the utilization of user and system resources. We feel that this problem, and its variations, are very relevant in the study of the complexity of *agent-based systems*. Essentially, an agent-based system is an environment within which an agent must successfully accomplish a task. Agents interact with the environment by performing actions and these actions can result in a change of state of the environment. Our resource-dependent digraph traversal problem can easily be viewed as an agent-based system, and we intend to investigate exactly how the study of program schemes and logics can impact upon that of agent-based systems in a future paper.

Acknowledgement We are very grateful to the insightful comments of a referee which significantly improved the clarity and rigour of this paper.

References

- [1] S. Abiteboul and V. Vianu, Procedural and declarative database update languages, *Proc. of ACM Symp. on Principles of Database Systems*, ACM Press (1988) 240–250.
- [2] S. Abiteboul and V. Vianu, Fixpoint extensions of first-order logic and Datalog-like languages, *Proc. of 4th Ann. IEEE Symp. on Logic in Computer Science*, IEEE Press (1989) 71–79.
- [3] S. Abiteboul and V. Vianu, Procedural languages for database queries and updates, *Journal of Computer and System Sciences* **41** (1990) 181–229.
- [4] S. Abiteboul and V. Vianu, Generic computation and its complexity, *Proceedings of the 23rd Ann. ACM Symp. on Theory of Computing*, ACM Press (1991) 209–219.
- [5] S. Abiteboul and V. Vianu, Computing with first-order logic, *Journal of Computer and System Sciences* **50** (1995) 309–335.
- [6] S. Abiteboul, M.Y. Vardi and V. Vianu, Fixpoint logics, relational machines and computational complexity, *Journal of the Association for Computing Machinery* **44** (1997) 30–56.
- [7] A.A. Arratia-Quesada, S.R. Chauhan and I.A. Stewart, Hierarchies in classes of program schemes, *Journal of Logic and Computation* **9** (1999) 915–957.
- [8] A.K. Chandra and D. Harel, Structure and complexity of relational queries, *Journal of Computer and System Sciences* **25** (1982) 99–128.
- [9] A.K. Chandra and D. Harel, Horn clause queries and generalizations, *Journal of Logic Programming* **2** (1985) 1–15.
- [10] R. Constable and D. Gries, On classes of program schemata, *SIAM Journal of Computing* **1** (1972) 66–118.

- [11] A. Dawar, A restricted second-order logic for finite structures, *Information and Computation* **143** (1998) 154–174.
- [12] H.D. Ebbinghaus and J. Flum, *Finite Model Theory*, Springer-Verlag (1995).
- [13] J. Esparza and M. Nielsen, Decidability issues for Petri nets – a survey, *Journal of Information Processing and Cybernetics* **30** (1994) 143–160.
- [14] R. Fagin, Generalized first-order spectra and polynomial-time recognizable sets, in: *Complexity of Computation* (ed. R.M. Karp), *SIAM-AMS Proceedings* **7** (1974) 43–73.
- [15] H. Friedman, Algorithmic procedures, generalized Turing algorithms and elementary recursion theory, in: *Logic Colloquium 1969* (ed. R.O. Gandy, C.M.E. Yates), North-Holland (1971) 361–390.
- [16] M. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman (1979).
- [17] R.L. Gault and I.A. Stewart, An infinite hierarchy in a class of polynomial-time program schemes, *Theory of Computing Systems* **39** (2006) 753–783.
- [18] A. van Gelder, K.A. Ross and J.S. Schlipf, The well-founded semantics for general logic programs, *Proc. of ACM Symp. on Principles of Database Systems*, ACM Press (1988) 221–230.
- [19] G. Gottlob, Relativized logspace and generalized quantifiers over finite ordered structures, *Journal of Symbolic Logic* **62** (1997) 545–574.
- [20] Y. Gurevich and S. Shelah, Fixed-point extensions of first-order logic, *Annals of Pure and Applied Logic* **32** (1986) 265–280.
- [21] D. Harel and D. Peleg, On static logics, dynamic logics, and complexity classes, *Information and Control* **60** (1984) 86–102.
- [22] N. Immerman, Relational queries computable in polynomial time, *Information and control* **69** (1986) 86–104.
- [23] N. Immerman, Languages that capture complexity classes, *SIAM Journal of Computing* **16** (1987) 760–778.
- [24] N. Immerman, *Descriptive Complexity*, Springer-Verlag (1998).
- [25] N.D. Jones and S.S. Muchnik, Even simple programs are hard to analyze, *Journal of Association for Computing Machinery* **24** (1977) 338–350.
- [26] L. Libkin, *Elements of Finite Model Theory*, Springer-Verlag, Berlin (2004).
- [27] F. Neven, M. Otto, J. Tyszkiewicz and J. van den Bussche, Adding for-loops to first-order logic, *Information and Computation* **168** (2001) 156–186.
- [28] M. Otto, *Bounded Variable Logics and Counting*, *Lecture Notes in Logic Volume 9*, Springer-Verlag (1997).

- [29] M. Paterson and N. Hewitt, Comparative schematology, *Record of Project MAC Conf. on Concurrent Systems and Parallel Computation*, ACM Press (1970) 119–128.
- [30] V. Sazonov, Polynomial computability and recursivity in finite domains, *Elektronische Informationsverarbeitung und Kybernetik* **60** (1980) 319–323.
- [31] I.A. Stewart, Complete problems involving boolean labelled structures and projection translations, *Journal of Logic and Computation* **1** (1991) 861–882.
- [32] I.A. Stewart, Using the Hamiltonian path operator to capture NP, *Journal of Computer and System Sciences* **45** (1992) 127–151.
- [33] I.A. Stewart, Logical and schematic characterization of complexity classes, *Acta Informatica* **30** (1993) 61–87.
- [34] I.A. Stewart, Methods for proving completeness via logical translations, *Theoretical Computer Science* **118** (1993) 193–229.
- [35] I.A. Stewart, Logical characterizations of bounded query classes II: polynomial-time oracle machines, *Fundamenta Informaticae* **18** (1993) 93–105.
- [36] I.A. Stewart, Complete problems for monotone NP, *Theoretical Computer Science* **145** (1995) 147–157.
- [37] I.A. Stewart, Logics with zero-one laws that are not fragments of bounded-variable infinitary logic, *Mathematical Logic Quarterly* **41** (1997) 158–178.
- [38] I.A. Stewart, Program schemes, queues, the recursive spectrum and zero-one laws, *Proc. of 7th Ann. Int. Computing and Combinatorics Conference* (ed. J. Wang), Lecture Notes in Computer Science Vol. 2108, Springer-Verlag, Berlin (2001) 39–48.
- [39] I.A. Stewart, Program schemes, arrays, Lindström quantifiers and zero-one laws, *Theoretical Computer Science* **275** (2002) 283–310.
- [40] I.A. Stewart, Using program schemes to logically capture polynomial-time on certain classes of structures, *London Mathematical Society Journal of Computation and Mathematics* **6** (2003) 40–67.
- [41] J. Tiuryn and P. Urzyczyn, Some relationships between logics of programs and complexity theory, *Theoretical Computer Science* **60** (1988) 83–108.
- [42] M. Vardi, The complexity of relational query languages, *Proc. of 14th Ann. ACM Symp. on Theory of Computing*, ACM Press (1982) 137–146.
- [43] V. Vianu, Databases and finite-model theory, in *Descriptive Complexity and Finite Models* (ed. N. Immerman and P. Kolaitis), DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 31, American Mathematical Society (1996), 97–148.