Logical definability versus computational complexity: another equivalence

Iain A. Stewart*
Department of Mathematics and Computer Science,
University of Leicester, Leicester LE1 7RH, U.K.

Abstract

We define a class of program schemes, NPSB, as the union of an infinite hierarchy of classes of program schemes NPSB(1) $\subseteq$ NPSB(2) $\subseteq$ ..., where our program schemes are built around 'high-level' programming constructs such as arrays, while-loops, assignments, and non-determinism, and take finite structures as their inputs. Every program scheme of NPSB(i) is actually also a program scheme of an existing class of program schemes NPSA(i), with NPSA defined analogously to NPSB. It has previously been shown that the class of problems accepted by the program schemes of NPSA: is contained in **PSPACE**; can be realized as the class of problems definable by the sentences of a certain vectorized Lindström logic; and has a zero-one law. We prove here that the class of problems accepted by the program schemes of NPSB is contained within the complexity class $\mathbf{L}^{\mathbf{NP}}$ and can also be realized as the class of problems definable by the sentences of a certain vectorized Lindström logic; and we exhibit a problem from NPSB(1) that is not definable in bounded-variable infinitary logic (which is the source of many zero-one laws in finite model theory). Furthermore, we prove that the complexity-theoretic question of whether **NP** equals **PSPACE** is equivalent to the logical question of whether the classes of problems accepted by the program schemes of NPSB(1) and NPSA(1) are identical; and that these latter classes of problems are none other than **NP**$\cap\mathcal{EXT}$ and **PSPACE**$\cap\mathcal{EXT}$, respectively, where $\mathcal{EXT}$ is the class of all problems closed under extensions. A corollary is that the classes of problems **NP**$\cap\mathcal{EXT}$ and **PSPACE**$\cap\mathcal{EXT}$ can be captured by logics (in the sense of Gurevich).

1 Introduction

Program schemes are models of computation defined around 'high-level' programming constructs such as arrays, while-loops, assignments, non-determinism and so on, and which take finite structures (over some fixed signature) as their inputs. These finite

*Supported by EPSRC Grants GR/K 96564 and GR/M 12933.

structures need not come equipped with a linear ordering of their elements (as is essentially the case when inputs are presented to Turing machines). The variables of any program scheme take values from the universe of the input structure. In a recent series of papers, various classes of program schemes have been shown to have a very close relationship with some of the logics considered in finite model theory. For example, it was proven in [3] that a class of program schemes NPS, defined around while-loops, assignments, conditional instructions and non-determinism, accept precisely the same class of problems as is definable by the sentences of Immerman's *transitive closure logic* (see [11]); and if we allow these program schemes to have access to a stack then the resulting class of program schemes, NPSS, accepts the class of problems definable by the sentences of *path system logic* (which has previously been studied in, for example, [9, 13, 18]). In [21], it was proven that a class of program schemes NPSA, defined by allowing the program schemes of NPS to have access to arrays, accepts a class of problems realizable as the class of problems definable by the sentences of a certain vectorized Lindström logic; and that even though this class of problems has a zero-one law, there exist problems in this class not definable in bounded-variable infinitary logic $\mathcal{L}^{\omega}_{\infty\omega}$ (many of the logics of finite model theory which have zero-one laws do so because they are essentially fragments of $\mathcal{L}^{\omega}_{\infty\omega}$: see, for example, [5]). In the presence of a built-in successor relation (or, equivalently, on the class of ordered finite structures), our classes of program schemes NPS, NPSS and NPSA capture the complexity classes **NL**, **P** and **PSPACE**, respectively [11, 16, 18].

In this paper, we restrict the program schemes of NPSA, to obtain a class of program schemes NPSB, by forcing our program schemes to be, in a sense, 'write-once'. That is, there are two distinct constants, 0 and *max*, available in every program scheme so that initially every array element (and variable) has the value 0 and the only assignment instructions allowed which write to array elements are ones which set an array element to the value *max*. Hence, once an array element has been set to the value *max*, it stays at *max* thereafter. It turns out that restricting array assignments in this way drops us down from **PSPACE** to $\mathbf{L}^{\mathbf{NP}}$ (the complexity class, contained in the second level of the Polynomial Hierarchy, consisting of all those problems accepted by logspace deterministic oracle Turing machines with access to an oracle from **NP**) in that every problem definable by a program scheme of NPSB is in $\mathbf{L}^{\mathbf{NP}}$ (in the presence of a built-in successor relation, the class of program schemes NPSB captures the complexity class $\mathbf{L}^{\mathbf{NP}}$).

Our classes of program schemes NPSA and NPSB are actually unions of hierarchies of classes of program schemes, namely NPSA $= \cup_{i=1}^{\infty}$NPSA(i) and NPSB $= \cup_{i=1}^{\infty}$NPSB(i), where NPSB$(i) \subseteq$ NPSA(i), for all i. We show that the first levels of these hierarchies, NPSA(1) and NPSB(1), have equivalent characterizations as the classes of problems **PSPACE**$\cap\mathcal{EXT}$ and **NP**$\cap\mathcal{EXT}$, respectively, where $\mathcal{EXT}$ is the class of all problems that are closed under extensions. This fact enables us to obtain a logical equivalence of an open complexity-theoretic problem: we prove that **NP** = **PSPACE** if, and only if, the classes of problems accepted by the program schemes of NPSB(1) and NPSA(1), namely **PSPACE**$\cap\mathcal{EXT}$ and **NP**$\cap\mathcal{EXT}$, are identical. Note that the classes of problems accepted by the program schemes of NPSB(1) and NPSA(1) have a zero-one law; in fact, all such problems are closed under extensions and so they have what might be termed a 'one-law'! We remark that the above equivalence is indeed a 'logical' equivalence in that there are logics,

in the sense of Gurevich [10], defining the classes of problems **PSPACE**$\cap\mathcal{EXT}$ and **NP**$\cap\mathcal{EXT}$ (these logics are actually sub-classes of the classes of program schemes NPSA(1) and NPSB(1): as to why the classes of program schemes NPSA(1) and NPSB(1) might not be logics, the reader is referred to Section 3). The fact that the classes of problems **PSPACE**$\cap\mathcal{EXT}$ and **NP**$\cap\mathcal{EXT}$ can be captured by logics is interesting in its own right.

Our logical equivalence result, mentioned in the previous paragraph, should be compared with Abiteboul and Vianu's seminal result, in [1], that **P** = **PSPACE** if, and only if, inductive fixed-point logic has the same expressibility as partial fixed-point logic (on the class of all finite structures). Again, note that the classes of problems definable by the sentences of inductive fixed-point logic and partial fixed-point logic have zero-one laws (essentially because inductive fixed-point logic and partial fixed-point logic can be regarded as fragments of $\mathcal{L}^{\omega}_{\infty\omega}$). Abiteboul, Vardi and Vianu [2] and Dawar [4] subsequently related numerous open complexity-theoretic questions, involving complexity classes ranging from **P** up to **EXPTIME**, with logical ones where the logics involved are essentially all fragments of $\mathcal{L}^{\omega}_{\infty\omega}$. Our result is very different in that it involves logics not realizable as fragments of $\mathcal{L}^{\omega}_{\infty\omega}$ and which have a 'one-law' as opposed to a zero-one law.

This paper is organised as follows. In the next section, we define basic results and notions from finite model theory (and, of particular relevance to this paper, we detail Gurevich's definition of what it means to be a 'logic' in the context of capturing complexity classes). In Section 3, we define our classes of program schemes, prove that (some of) our classes of problems can be captured by logics and also prove our Abiteboul-Vianu-style characterization. In Section 4, we prove a strong completeness result for the class of problems accepted by the program schemes of NPSB(1) before exhibiting a vectorized Lindström logic capturing the class of problems accepted by the program schemes of NPSB. We give our conclusions in Section 5.

2 Preliminaries

The main reference texts for the basic concepts, notions and results of finite model theory are [5, 12] and it is to these books that we refer the reader for additional information. However, we give the definitions relating to this paper in some detail below as it is often the case that more generality is required than in [5, 12], and we also need some notions not defined in those texts.

Ordinarily, a *signature* σ is a tuple $\langle R_1, \ldots, R_r, C_1, \ldots, C_c\rangle$, 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, FO(σ), consists of those formulae built from atomic formulae over σ using $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$; and FO $= \cup\{\mathrm{FO}(\sigma) : \sigma \text{ is some signature}\}$.

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 class of all finite structures of size at least 2 over the signature σ is denoted

STRUCT(σ). A *problem* over some signature σ consists of a sub-class of STRUCT(σ) 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 (and classes of program schemes). 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 (more precisely, an encoding of any first-order definable problem) can be accepted by a logspace deterministic Turing machine yet there are (encodings of) problems in the complexity class **L** 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 $\sigma = \langle R_1, \ldots, R_r, C_1, \ldots, C_c \rangle$, as above. The logic $(\pm\Omega)^*$[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 $\psi_1(\mathbf{x}_1, \mathbf{y}), \ldots, \psi_r(\mathbf{x}_r, \mathbf{y})$ are formulae of $(\pm\Omega)^*$[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 $\mathbf{x}_1, \ldots, \mathbf{x}_r$; and
  - all free variables of any ψ_i are contained in either $\mathbf{x}_i$ or $\mathbf{y}$.
- Suppose that $\mathbf{d}_1, \ldots, \mathbf{d}_c$ are k-tuples of variables and constant symbols (which need not be distinct).
- Then
$$\Omega[\lambda \mathbf{x}_1 \psi(\mathbf{x}_1, \mathbf{y}), \ldots, \mathbf{x}_r \psi_r(\mathbf{x}_r, \mathbf{y})](\mathbf{d}_1, \ldots, \mathbf{d}_c)$$
is a formula of $(\pm\Omega)^*$[FO] whose free variables are the variables of $\mathbf{y}$ together with any other variables appearing in $\mathbf{d}_1, \ldots, \mathbf{d}_c$.

If Φ is a sentence of the form $\Omega[\lambda \mathbf{x}_1 \psi(\mathbf{x}_1), \ldots, \mathbf{x}_r \psi_r(\mathbf{x}_r)](\mathbf{d}_1, \ldots, \mathbf{d}_c)$, 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 $\mathbf{d}_1, \ldots, \mathbf{d}_c$, which are only there if there are constant symbols in σ, consist entirely of constant symbols of σ'). First, we build a σ-structure $\Phi(\mathcal{A})$.

- 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 $(\pm\Omega)^*[\mathrm{FO}]$ *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 $\Omega^*[\mathrm{FO}]$ (the *positive* fragment of $(\pm\Omega)^*[\mathrm{FO}]$); and further the formulae are such that there are no nestings of the operator Ω, namely $\Omega^1[\mathrm{FO}]$ (the *positive unnested* fragment of $(\pm\Omega)^*[\mathrm{FO}]$).

It can be the case that (a fragment of) a vectorized Lindström logic $(\pm\Omega)^*[\mathrm{FO}]$ 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) $(\pm\Omega)^*[\mathrm{FO}]$ can be defined by a sentence of the form $\Omega[\lambda\mathbf{x}_1\psi_1(\mathbf{x}_1), \ldots, \mathbf{x}_r\psi_r(\mathbf{x}_r)](\mathbf{d}_1, \ldots, \mathbf{d}_c)$, 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) $(\pm\Omega)^*[\mathrm{FO}]$ 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, [8, 11, 14, 15] 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 $(\pm\Omega)^*[\mathrm{FO}]$. To say that this logic has a *built-in successor relation*, which we denote by $(\pm\Omega)^*[\mathrm{FO}_s]$, 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 $\{(0, u_1), (u_1, u_2), \ldots, (u_{n-2}, max)\}$, where the elements of $\{0, u_1, u_2, \ldots, u_{n-2}, max\}$ are distinct. However, there is a further semantic stipulation on the sentences of $(\pm\Omega)^*[\mathrm{FO}_s]$: 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 $(\pm\mathrm{TC})^*[\mathrm{FO}_s]$ (as it is invariant with respect to *succ*):

$$\mathrm{TC}[\lambda(x_1, x_2), (y_1, y_2)(x_1 = 0 \wedge y_1 = max \wedge succ(x_2, y_2)) \\ \vee (x_1 = max \wedge y_1 = 0 \wedge succ(x_2, y_2))](0, 0, max, max),$$

and it defines the problem over the empty signature consisting of those structures of even size. (Note that in [5], for example, 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. Nevertheless, the two approaches essentially amount to the same thing. 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* (see, for example, [5]). 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' $(\pm\Omega)^*[\mathrm{FO}_s]$ 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 **P** (or indeed any complexity class contained in **NP**, where by 'contained in' we really mean 'contained in but expected to be different from'; as **NP** itself can be captured by a logic, one such being existential second-order logic). Of course, in order to make sense of this question, one needs to be more precise as to what one means by a 'logic'. Gurevich [10] has proposed the following definition for a logic (in the context of capturing complexity classes) and this definition (which has been widely adopted) is as follows. A *logic* L is given by a pair of functions (Sen, Sat) satisfying the following conditions. The function Sen associates with every signature σ a recursive set $Sen(\sigma)$ whose elements are called *L-sentences over σ*. The function Sat associates with every signature a recursive relation $Sat_\sigma(\mathcal{A}, \varphi)$, where $\mathcal{A}$ is a σ-structure and φ is a sentence of L. We say that $\mathcal{A}$ *satisfies* φ (and write $\mathcal{A} \models \varphi$) if $Sat_\sigma(\mathcal{A}, \varphi)$ holds. Furthermore, we require that $Sat_\sigma(\mathcal{A}, \varphi)$ if, and only if, $Sat_\sigma(\mathcal{B}, \varphi)$ when $\mathcal{A}$ and $\mathcal{B}$ are isomorphic. We shall return to this definition later when we define our classes of program schemes. (Note that, notwithstanding the above discussion, we still refer to $(\pm\Omega)^*[\mathrm{FO}_s]$ as a logic on the grounds of convenience.)

Returning to vectorized Lindström logics and normal forms, 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.

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

(think of a σ_2-structure $\mathcal{G}$ as encoding an undirected graph via: 'there is an edge (u, v) if, and only if, $E(u, v) \vee E(v, u)$ holds in $\mathcal{G}$').

Theorem 1 [19] *The complexity class* **NP** *is identical to the class of problems defined by the sentences of $CUB^*[FO_s]$; and any problem in* **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, $CUB^*[FO_s] = CUB^1[FO_s]$ and CUB is complete for **NP** *via quantifier-free first-order translations with successor.* □

Note that Theorem 1 subsumes the 'traditional' known complexity-theoretic result that CUB is complete for **NP** via logspace reductions (a result attributed to Chvàtal in [7]). We shall need Theorem 1 later on.

3 Program schemes

Program schemes are more 'computational' means for defining classes of problems than are logical formulae. A *program scheme* $\rho \in$ NPSA(1) involves a finite set $\{x_1, x_2, \ldots, x_k\}$ 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 $i \in \{1, 2, \ldots, k\}$ and where y is a variable from $\{x_1, x_2, \ldots, x_k\}$, 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 `:=` $A[y_1, y_2, \ldots, y_d]$' or '$A[y_1, y_2, \ldots, y_d]$ `:=` y_0', for some $i \in \{1, 2, \ldots, k\}$, where each y_j is a variable from $\{x_1, x_2, \ldots, x_k\}$, 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 *arity* d;
- a *guess instruction* of the form '`guess` x_i', where $i \in \{1, 2, \ldots, k\}$; or
- a *while instruction* of the form '`while` φ `do` $\alpha_1; \alpha_2; \ldots; \alpha_q$ `od`', where φ is a quantifier-free formula of FO($\sigma \cup \{0, max\}$), whose free variables are from $\{x_1, x_2, \ldots, x_k\}$, and where each of $\alpha_1, \alpha_2, \ldots, \alpha_q$ 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`$(x_1, x_2, \ldots, x_l)$' and the last instruction is '`output` $(x_1, x_2, \ldots, x_l)$', for some l where $1 \leq l \leq k$. The variables $x_1, x_2, \ldots, x_l$ are the *input-output variables* of ρ, the variables $x_{l+1}, x_{l+2}, \ldots, x_k$ 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 $\rho \in$ 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:

- any array A of arity d is indexed by d-tuples of elements of $|\mathcal{A}|$ and every array element takes a value from $|\mathcal{A}|$;
- 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, [16]): henceforth, we shall assume that these instructions are at our disposal.

We want the classes 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. Consequently, not every program scheme, built as above, is well-formed. This is analogous to how we build two constant symbols (or a successor relation) into a logic as described earlier. Furthermore, we can build a successor relation into the program schemes of NPSA(1) so as to obtain the class of program schemes $\mathrm{NPSA}_s(1)$. As with our logics, we write NPSA(1) and $\mathrm{NPSA}_s(1)$ to also denote the class of problems accepted by the program schemes of NPSA(1) and $\mathrm{NPSA}_s(1)$, respectively. It was proven in [16] that a problem is in the complexity class **PSPACE** if, and only if, it is in $\mathrm{NPSA}_s(1)$.

Henceforth, we think of the program schemes of NPSA(1) 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 the number of the instruction about to be executed, a value for each variable and values for all array elements. A *partial ID* consists of just the number of the instruction about to be executed and a value for each variable. 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.

We now extend the class of program schemes NPSA(1).

Definition 2 Let σ be some signature. For some $m \geq 1$, let the program scheme $\rho \in \mathrm{NPSA}(2m-1)$ be over the signature σ and involve the variables $x_1, x_2, \ldots, x_k$. Suppose that the variables $x_1, x_2, \ldots, x_l$ are the input-output variables of ρ, the variables $x_{l+1}, x_{l+2}, \ldots, x_{l+s}$ are the free variables, and the remaining variables are the bound variables (note that if $m = 1$ then ρ has no bound variables but that this may not be the case if $m > 1$). Let $x_{i_1}, x_{i_2}, \ldots, x_{i_p}$ be free variables of ρ, for some p such that $1 \leq p \leq s$. Then

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

is a program scheme of $\mathrm{NPSA}(2m)$, which we denote by ρ', with no input-output variables, with free variables those of $\{x_{l+1}, x_{l+2}, \ldots, x_{l+s}\} \setminus \{x_{i_1}, x_{i_2}, \ldots, x_{i_p}\}$ and with the remaining variables of $\{x_1, x_2, \ldots, x_k\}$ 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}), $\mathcal{A}'' \models \rho$. □

Note that the different computations of ρ on expansions $\mathcal{A}''$ of $\mathcal{A}'$, in Definition 2, are all such that all arrays are initialised to 0. Also, note that for ρ' to be well-formed, we require that ρ is well-formed.

Definition 3 Let σ be some signature. A program scheme $\rho' \in \mathrm{NPSA}(2m-1)$, for some $m \geq 2$, over the signature σ and involving the variables of $\{x_1, x_2, \ldots, x_k\}$, is formed exactly as are the program schemes of NPSA(1), with the input-output and free variables defined accordingly, except that the test in some while instruction is a program scheme $\rho \in \mathrm{NPSA}(2m-2)$ whose free and bound variables are all from $\{x_1, x_2, \ldots, x_k\}$ (note that ρ has no input-output variables). However, there are further stipulations:

- all free variables in any test $\rho \in \mathrm{NPSA}(2m-2)$ in any while instruction are input-output or free variables of ρ';
- the bound variables of ρ' consist of all bound variables of any test $\rho \in \mathrm{NPSA}(2m-2)$ in any while instruction (and no bound variable is ever an input-output or free variable of ρ); and
- this accounts for all variables of $\{x_1, x_2, \ldots, x_k\}$.

Of course, any free variable of ρ' never appears on the left-hand side of an assignment instruction or in a guess instruction.

A program scheme $\rho' \in \mathrm{NPSA}(2m-1)$ takes expansions $\mathcal{A}'$ of σ-structures $\mathcal{A}$ by adjoining s constants as input, where s is the number of free variables, and computes on $\mathcal{A}'$ in the obvious way except that when some while instruction is encountered, the test, which is a program scheme $\rho \in \mathrm{NPSA}(2m-2)$, 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, all arrays in ρ are initialised to 0 and when the test has been evaluated the computation of ρ' resumes accordingly with its arrays having exactly the same values as they had immediately before the test was evaluated. □

Again, note that for ρ' to be well-formed, we require that ρ is well-formed.

In a program scheme such as ρ' in Definition 3, the only information which can be 'passed' to a test evaluation is the current values of the relevant input-output or free variables. Arrays can not be used to pass information across.

A simple analysis yields that we can build the usual 'if' and 'if-then-else' instructions in the program schemes of NPSA(m), for all odd $m \geq 1$. Indeed, henceforth we assume that these instructions are at our disposal.

Amongst the results proven in [21] are that the class of problems accepted by the program schemes of NPSA: has a zero-one law; is realizable as the class of problems defined by the sentences of a certain vectorized Lindström logic; contains problems not definable in bounded-variable infinitary logic; and captures **PSPACE** in the presence of a built-in successor relation (the reader is referred to, for example, [5] for more on bounded-variable infinitary logic and zero-one laws).

We can now define the class of program schemes NPSB(i) within NPSA(i), for any $i = 1, 2, \ldots$, as being those programs schemes where all assignment instructions with an array symbol involved in the term on the left-hand side of the assignment

are such that the term on the right-hand side of the assignment is max. That is, an array element can only be set to the value max (after starting off at the value 0) and once set, it stays at the value max thereafter. The class of program schemes NPSB is defined as $\cup_{i=1}^{\infty}$NPSB(i).

Let us illustrate how a program scheme of NPSB(1) accepts a problem with the following lemma.

Lemma 4 *The problem CUB can be accepted by a program scheme of NPSB*(1).

Proof It was shown in [21] that CUB is in NPSA(1): however, the program scheme used there to accept CUB is not in NPSB(1). Nevertheless, the basic approach can be amended to yield a program scheme of 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 array symbols, B_1 and B_2, of arity 3 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 $B_1[0, 0, u_1] = max$ and $B_2[0, 0, v_1] = max$. 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 $B_1[u_1, v_1, u_2] = max$ and $B_2[u_1, v_1, v_2] = max$: otherwise, we set $B_1[u_1, v_1, max] = max$ and $B_2[u_1, v_1, max] = max$ and stop guessing. Assuming that we have not yet stopped guessing, we guess an edge (u_3, v_3), 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) and (u_2, v_2). If so then we set $B_1[u_2, v_2, u_3] = max$ and $B_2[u_2, v_2, v_3] = max$: otherwise, we set $B_1[u_2, v_2, max] = max$ and $B_2[u_2, v_2, max] = max$ 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 NPSB(1): hence, the result follows. □

Using Theorem 1, and allowing access to a built-in successor relation, we can easily modify the program scheme implicit in the proof of Lemma 4 so that it accepts any given problem in **NP** (essentially, we have simply to 'vectorize' the program scheme). Conversely, any problem in NPSB$_s$(1) is in **NP**. Hence, we have the following corollary.

Corollary 5 *As classes of problems,* **NP** = *NPSB*$_s$(1). □

Now for an important remark. As the definition of our class of program schemes NPSA(1), for example, stands, we do not know whether the program schemes in this class form a logic in Gurevich's sense. However, there is a logic defining exactly the problems accepted by the program schemes of NPSA(1).

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 $\mathcal{EXT}$ be the class of all problems that are closed under extensions.

Lemma 6 *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, $\mathcal{B} \in \Omega$. □

Theorem 7 *There is a logic capturing the class of problems NPSA*(1).

Proof Let ρ be any program scheme built as are the program schemes of NPSA(1) but so that ρ might not be well-formed and so that ρ has no free variables. We shall build a well-formed program scheme $f(\rho) \in$ NPSA(1). Let $\mathbf{x}$ be the input-output variables of ρ. The program scheme $f(\rho)$ beings with code which guesses two distinct values for the variables z_0 and z_m which we assume do not occur in $\mathbf{x}$. Moreover, we ensure that the constant symbols 0 and max are not used in this portion of code. Next, we include code to guess a list of distinct elements of the form $z_0, w_1, w_2, \ldots, w_k, z_m$, for some $k \geq 0$, of the input structure which we store using a new array symbol B of arity 1 which we assume does not appear in ρ. These values are stored via: $B[z_0] = w_1$; $B[w_1] := w_2$; $\ldots$; $B[w_k] = z_m$. Again, we ensure that we do not use the constant symbols 0 and max in this fragment of code and that the variables z_0 and z_m do not appear in a guess instruction or on the left-hand side of an assignment instruction (of course, we can use the fact that the value of z_0 is different from the value of z_m in this fragment of code).

Next, we include in $f(\rho)$ an amended version of ρ where we strip away the input- and output-instructions, and we replace every occurrence of the constant symbol 0 or max with the variable z_0 or z_m, respectively. We also 'guard' every guess instruction as follows. We replace every guess instruction `guess` x_i with the following fragment of code (where s and t are new variables):

```
guess x_i
s := z_0
t := z_0
while s = z_0 do
   if x_i = t then
      s := z_m
   else
      if t ≠ z_m then
         t := B[t]
         fi
      fi
   od
```

Consequently, we are restricting guesses to the list of elements held in the array B. Finally, we check to see whether the variables of $\mathbf{x}$ are set all set at z_m and if they are then we signal acceptance by setting every variable to max; otherwise we set every variable to 0 (note that this is the only place where either of the the constant symbols 0 and max appear in $f(\rho)$).

An immediate observation is that regardless of ρ, the program scheme $f(\rho)$ is a well-formed program scheme of NPSA(1). Suppose that ρ is well-formed and that $\mathcal{A} \models \rho$. Then $\mathcal{A} \models f(\rho)$ also (we guess the list of elements held in B to be the whole of the universe $|\mathcal{A}|$ and mirror an accepting computation of ρ on input $\mathcal{A}$). Conversely, suppose that ρ is well-formed and that $\mathcal{A} \models f(\rho)$. Then there exists a substructure $\mathcal{B}$ of $\mathcal{A}$ such that $\mathcal{B} \models \rho$. However, by Lemma 6, the problem accepted by the program scheme ρ is closed under extensions; and so $\mathcal{A} \models \rho$. Thus, if ρ is well-formed then ρ and $f(\rho)$ accept exactly the same problem.

The class of program schemes

$$\{f(\rho) \quad : \quad \rho \text{ is built as are the program schemes of NPSA(1) but so that } \rho \text{ might not be well-formed}\}$$

is clearly a logic, in Gurevich's sense, and so the result follows. □

We can amend the program scheme $f(\rho)$ of the proof of Theorem 7 so that if ρ is built as are the program schemes of NPSB(1) then $f(\rho)$ is also a program scheme of NPSB(1) as follows. We replace the array symbol B with an array symbol C of arity 2, and any assignment of the form $B[w_1] := w_2$, say, in the program scheme $f(\rho)$ with an assignment $C[w_1, w_2] := z_m$ (together with some consequent adjustments in tests in which B appears). Hence, we obtain the following immediate corollary.

Corollary 8 *There is a logic capturing the class of problems NPSB*(1). □

We now equate the complexity-theoretic question of whether the complexity classes **NP** and **PSPACE** are equal with that of whether the classes of problems NPSB(1) and NPSA(1) are equal (we emphasise that we are not assuming the existence of a built-in successor relation).

Theorem 9 $\mathbf{PSPACE} \cap \mathcal{EXT} = NPSA(1)$ *and* $\mathbf{NP} \cap \mathcal{EXT} = NPSB(1)$.

Proof Let Ω be some problem in $\mathbf{PSPACE} \cap \mathcal{EXT}$. By [16], there exists a program scheme $\rho \in \text{NPSA}_s(1)$ accepting Ω. Modify ρ to obtain the program scheme $f(\rho) \in$ NPSA(1) as follows. In $f(\rho)$, begin by guessing a successor relation; that is, when $\mathcal{A}$ is some input structure, guess elements $u_1, u_2, \ldots, u_m \in |\mathcal{A}|$ so that

$$B[0] = u_1, B[u_1] = u_2, \ldots, B[u_m] = max,$$

where B is a new array symbol and where the elements of $\{0, u_1, u_2, \ldots, u_m, max\}$ 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 $y = B[x]$, and guard every guess instruction as in the proof of Theorem 7. We need to show that acceptance by the program scheme $f(\rho)$ is invariant with respect to 0 and max and that $f(\rho)$ 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 $f(\rho)$ on input $\mathcal{A}$ where the guessed successor relation is $succ'$. Then there exists a computation of $f(\rho)$ mirroring any accepting computation of ρ on input $\mathcal{A}$ with this particular successor relation. That is, $\mathcal{A}$ is accepted by $f(\rho)$ and acceptance does not depend upon the chosen constants 0 and *max*.

Conversely, suppose that $\mathcal{A}$ is accepted by $f(\rho)$, with respect to some guessed successor relation, call it $succ'$ (whose domain need not be all of $|\mathcal{A}|$), with minimal and maximal elements $0'$ and max'. Let $\mathcal{B}$ be the sub-structure of $\mathcal{A}$ whose domain is the domain of this successor relation. 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 $f(\rho)$ and acceptance does not depend upon the chosen constants 0 and *max*. Thus, acceptance by $f(\rho)$ is invariant with respect to 0 and *max*; and $\mathbf{PSPACE} \cap \mathcal{EXT} \subseteq$ NPSA(1). The fact that every problem in NPSA(1) can be solved by a polynomial-space algorithm is straight-forward; and every problem in NPSA(1) is closed under extensions by Lemma 6.

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

$$B[0, u_1] = max, B[u_1, u_2] = max, \ldots, B[u_m, max] = max,$$

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

Corollary 10 $\mathbf{NP} = \mathbf{PSPACE}$ *if, and only if,* $NPSA(1) = NPSB(1)$.

Proof If $\mathbf{NP} = \mathbf{PSPACE}$ then $\mathbf{NP} \cap \mathcal{EXT} = \mathbf{PSPACE} \cap \mathcal{EXT}$; and so NPSA(1) = NPSB(1) by Theorem 9. Conversely, if NPSA(1) = NPSB(1) then $\mathrm{NPSA}_s(1) = \mathrm{NPSB}_s(1)$; and so $\mathbf{NP} = \mathbf{PSPACE}$ by [16] and Corollary 5. □

Corollary 10 is somewhat surprising given that every problem in NPSA(1) (and so NPSB(1)) is closed under extensions and so has a zero-one law (in fact, a 'one-law'). Note that a class of problems each of which has a zero-one law can not contain, for example, the (computationally trivial) problem consisting of all those structures of even size.

It is also the case that CUB $\in$ NSPB(1), by Lemma 4, and that CUB is not definable in bounded-variable infinitary logic, by [20]. Consequently, NPSB(1) (and so NPSA(1)) can not be realized as a fragment of bounded-variable infinitary logic, unlike the logics in [1, 2, 4] mentioned in the Introduction.

4 Partitioned Petri nets

In this section, we show that the class of problems NPSB(1) has a complete problem via quantifier-free first-order translations with two built-in constants. The particular complete problem is somewhat contrived and peculiar in that it involves Petri nets whose places are partitioned into disjoint sets (these sets govern the types of transition that can exist within the Petri net). Nevertheless, the fact that our problem is complete ultimately enables us to obtain a characterization of the class of problems NPSB as those problems defined by the sentences of a vectorized Lindström logic whose associated operator corresponds to our complete problem.

Our notion of a Petri net is as usual and the reader is referred to, for example, [6] where basic Petri net notions and concepts are defined (this reference also gives details of numerous complexity-theoretic results concerning fundamental problems in Petri nets).

Definition 11 Define $\sigma_b = \langle P, Q, M, T_1, T_2, T_3, C, D \rangle$ where P, Q, M, T_1, T_2 and T_3 are relation symbols of arities 1, 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

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

- the relation T_2 as describing the set of transitions

$$\{(\{u, w\}, \{v, w\}) : u, v \in P, w \notin P, T_2(u, v, w) \text{ holds}\};$$

  and

- the relation T_3 as describing the set of transitions

$$\{(\{u, w\}, \{v, w'\}) : u, v \in P, w, w' \notin P, w \in Q, w' \notin Q, T_3(u, v, w, w') \text{ holds}\}.$$

Furthermore, we can think of the relation M and the constant C as describing an initial marking of our Petri net via: there is one token on place u if, and only if, $(M(u) \wedge \neg P(u) \wedge Q(u)) \vee u = C$ holds. We define the problem Ω_b as

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

Note that the Petri nets obtained from σ_b-structures are severely restricted. □

Note that the transitions encoded within a σ_b-structure $\mathcal{P}$ are of one of four types, as depicted in Fig. 1. Note also 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. We shall discuss why our problem Ω_b is as it is later on. The proof of the following theorem is similar to those in [21].

Theorem 12 *There is a quantifier-free first-order translation with two built-in 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 two built-in 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 arity $d \geq 1$. We assume that ρ has no free variables and that the input-output variables of ρ are $x_1, x_2, \ldots, x_k$.

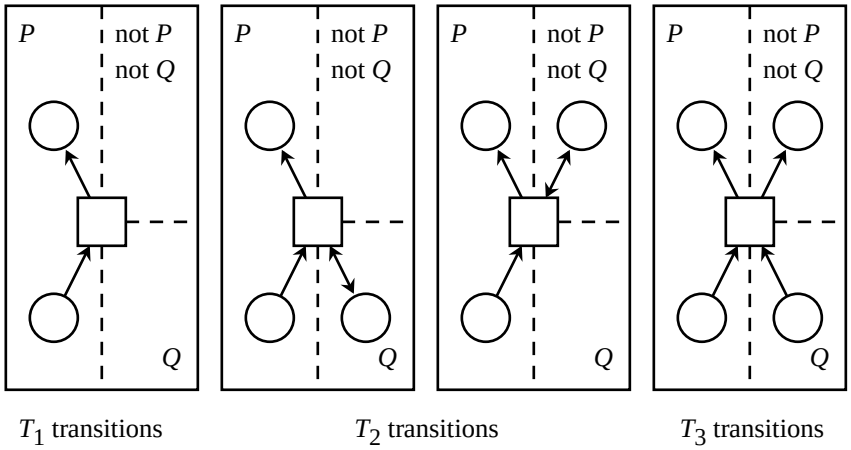

Figure 1. The different types of transitions.

Let $\mathcal{A}$ be a σ-structure of size $n \geq 2$. An element $\mathbf{u} = (u_0, u_1, \ldots, u_k)$ of $\{1, 2, \ldots, l\} \times |\mathcal{A}|^k$ 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 $x_1, x_2, \ldots, x_k$ currently have the values $u_1, u_2, \ldots, u_k$, respectively. Henceforth, we identify partial IDs of ρ and the elements of $\{1, 2, \ldots, l\} \times |\mathcal{A}|^k$. A (*full*) *ID* is a partial ID augmented with a valuation on all array elements.

We now build a Petri net $\mathcal{P}$, as in Definition 11, using ρ and $\mathcal{A}$. Our Petri net $\mathcal{P}$ has a set of places consisting of the set $\{1, 2, \ldots, l\} \times |\mathcal{A}|^k$ 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

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

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

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.

Suppose that the instruction u_0 is of the form $x_j := B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}]$ 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 $B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}]$, which is 0 or max, 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 $(\{\mathbf{u}, (u_{i_1}, u_{i_2}, \ldots, u_{i_d})_0\}, \{\mathbf{v}, (u_{i_1}, u_{i_2}, \ldots, u_{i_d})_0\})$ and $(\{\mathbf{u}, (u_{i_1}, u_{i_2}, \ldots, u_{i_d})_m\}, \{\mathbf{v}, (u_{i_1}, u_{i_2}, \ldots, u_{i_d})_m\})$ are in T_2.

Suppose that the instruction u_0 is of the form $B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}] := max$ 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 $(\{\mathbf{u}, (u_{i_1}, u_{i_2}, \ldots, u_{i_d})_m\}, \{\mathbf{v}, (u_{i_1}, u_{i_2}, \ldots, u_{i_d})_m\})$ is in T_2 and the transition $(\{\mathbf{u}, (u_{i_1}, u_{i_2}, \ldots, u_{i_d})_0\}, \{\mathbf{v}, (u_{i_1}, u_{i_2}, \ldots, u_{i_d})_m\})$ is in T_3.

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

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). Consequently, in order for the result to follow we need to show that $\rho \models \mathcal{A}$ if, and only if, $\mathcal{P} \in \Omega_b$; and that $\Omega_b \in$ NPSB(1).

Suppose that $\rho \models \mathcal{A}$. Then there is a sequence s 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 s to the next in one step. We can mirror any ID with a marking of our Petri net $\mathcal{P}$ as follows. If the ID consists of the partial ID $\mathbf{u} \in \{1, 2, \ldots, l\} \times |\mathcal{A}|^k$ together with some valuation on the array B then the place $\mathbf{u}$ is marked with one token as are the places of

$$\{\mathbf{w}_0 : \mathbf{w} \in |\mathcal{A}|^d, B[\mathbf{w}] = 0\} \cup \{\mathbf{w}_m : \mathbf{w} \in |\mathcal{A}|^d, B[\mathbf{w}] = max\}.$$

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. Hence, $\mathcal{A} \models \rho$ if, and only if, $\mathcal{P} \in \Omega_b$.

All that remains is to show that $\Omega_b \in$ 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 place; and we need to cater for this event when we simulate a computation in $\mathcal{P}$ in 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 at most one token, call it t, on a place p for which $P(p)$ holds and there is at most one token on every place p for which $\neg P(p) \wedge Q(p)$ holds. 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 p for which $P(p)$ holds, and we can imagine every other token either staying where it is, after some transition, or being moved from a place p for which $\neg P(p) \wedge Q(p)$ holds to a place q for which $\neg P(q) \wedge \neg Q(p)$ holds, 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 place p for which $\neg P(p) \wedge \neg Q(p)$ holds but only whether there is at least one token on p. This obviates the need to count tokens. As regards our second

difficulty, in order to decide whether (at least) one token lies on some place p for which $\neg P(p) \wedge \neg Q(p)$ holds, we simply need to have a dedicated array B_1, of arity 1, so that whenever a token is placed on such a place p then $B_1[p]$ is set at max: once $B_1[p]$ has been set to max we know that there will be a token on that place thereafter. In order to decide whether (at least) one token lies on some place p for which $\neg P(p) \wedge Q(p)$ holds, we use the array B_2, of arity 1, to register when the token originally on the place p (if there was one) is first moved from p by setting $B_2[p]$ equal to max at this point. Consequently, if we wish to know whether there is a token on such a place p, we test to see whether $M(p) \wedge B_2[p] = 0$ holds. Finally, we model the movement of the solitary token t (if it exists) 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. □

The following is immediate from Corollary 5.

Corollary 13 *The problem Ω_b is complete for* **NP** *via quantifier-free first-order translations with successor.* □

Theorem 12 tells us that any problem accepted by a program scheme of NPSB(1) can be described by a sentence of the form

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

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_M, 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 (moreover, the sentence is such that satisfiability is independent of 0 and max).

An analogous result to Theorem 12, except where the class of program schemes is NPSA(1) and the problem is Ω_a, was proven in [21]. The problem Ω_a is defined as follows.

Definition 14 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 inital 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. □

Theorem 15 [21] *There is a quantifier-free first-order translation with two built-in constants from any problem in NPSA(1) to the problem Ω_a. Hence, Ω_a is complete for NPSA(1) via quantifier-free first-order translations with two built-in constants, and also for* **PSPACE** *via quantifier-free first-order translations with successor.* □

Corollary 16 *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 two built-in constants.*

Proof Corollary 10 yields (*a*) ⇔ (*b*). Theorems 12 and 15 yield (*b*) ⇔ (*d*).

Suppose that (*c*) holds. Then $\Omega_a \in \Omega_b^1$[FO] and so Ω_a is in **NP**. But Ω_a is **PSPACE**-complete, by Theorem 15, and so (*a*) holds. Conversely, suppose that (*d*) holds. Then Ω_a can be defined by a sentence φ of Ω_b^1[FO] with two built-in constants. However, if we replace 0 and *max* in φ with two new variables y and z, respectively, to get φ', then the sentence $\exists y \exists z(y \neq z \wedge \varphi')$ of Ω_b^1[FO] defines Ω_a and there are no built-in constants. Hence, (*c*) holds. □

It is interesting to compare Corollary 16 with Abiteboul and Vianu's result mentioned in the Introduction. Of course, their result is for **P** and **PSPACE** whilst ours is for **NP** and **PSPACE** but there are other interesting differences. Their result, like similar results subsequently proven by Abiteboul, Vardi and Vianu [2] and Dawar [4], is for logics realizable as fragments of $\mathcal{L}_{\infty\omega}^{\omega}$, whereas our result is for logics not so realizable (recall, the classes of problems accepted by the program schemes of NPSA(1) and NPSB(1) can be captured by logics). Also, whilst any problem definable by a sentence of $\mathcal{L}_{\infty\omega}^{\omega}$ has a zero-one law, any problem of NPSA(1) is closed under extensions (and so has a '1-law'): of course, whilst any problem in Ω_a^1[FO] or Ω_b^1[FO] has a zero-one law, there are problems in these classes which are not closed under extensions. Finally, not only do we equate whether **NP** = **PSPACE** with the logical question of whether NPSB(1) = NPSA(1) but we also equate it with whether two complete problems for these classes are equivalent (with respect to quantifier-free first-order translations with two built-in constants). There is no such equivalence associated with any of the results of [1, 2, 4].

A simple induction, allied with the proof of Theorem 12, yields that the class of problems NPSB can be realized as the class of problems definable by the sentences of the vectorized Lindström logic $(\pm\Omega_b)^*$[FO] (the built-in constants 0 and *max* can be dispensed with as in the proof of Corollary 16).

Corollary 17 $NPSB = (\pm\Omega_b)^*[FO]$. □

Corollary 17 should be compared with the result, proven in [21], that NPSA = $(\pm\Omega_a)^*$[FO].

5 Conclusions

We would like to involve the full vectorized Lindström logics $(\pm\Omega_b)^*[\mathrm{FO}]$ and $(\pm\Omega_a)^*[\mathrm{FO}]$ (or even the positive fragments $\Omega_b^*[\mathrm{FO}]$ and $\Omega_a^*[\mathrm{FO}]$) in results similar to Corollary 16. Let us make some observations about these logics.

(a) For a logic $(\pm\Omega)^*[\mathrm{FO}]$, denote by $(\pm\Omega)^1[\mathrm{FO}]$ the fragment where the operator Ω does not appear nested in any formula of this fragment. By [17],

$$(\pm\Omega_b)^*[\mathrm{FO}_s] = (\pm\Omega_b)^1[\mathrm{FO}_s] = \mathbf{L}^{\mathbf{NP}}$$

(it was proven in [17] that an analogous result holds with the operator HP, corresponding to the problem of deciding whether there is a Hamiltonian path in a digraph from one vertex to another, replacing the operator Ω_b: our result then follows).

(b) By Corollary 16,

$$\text{if } \mathbf{NP} = \mathbf{PSPACE} \text{ then } (\pm\Omega_b)^*[\mathrm{FO}] = (\pm\Omega_a)^*[\mathrm{FO}].$$

However, we cannot deduce that if $\mathbf{NP} = \mathbf{PSPACE}$ then $\Omega_b^*[\mathrm{FO}] = \Omega_a^*[\mathrm{FO}]$. It may be the case that although Ω_a and Ω_b are equivalent via quantifier-free first-order translations with two built-in constants, the translation from Ω_a to Ω_b might negate some of the relation symbols of σ_a so that formulae involving positive nested applications of Ω_a translate to formulae of $(\pm\Omega_b)^*[\mathrm{FO}]$, not $\Omega_b^*[\mathrm{FO}]$.

(c) On the other hand,

$$\text{if } \Omega_b^*[\mathrm{FO}] = \Omega_a^*[\mathrm{FO}] \text{ then } \mathbf{NP} = \mathbf{PSPACE},$$

as it is not difficult to see that any problem definable in $\Omega_b^*[\mathrm{FO}]$ is in $\mathbf{NP}$; and

$$\text{if } (\pm\Omega_b)^*[\mathrm{FO}] = (\pm\Omega_a)^*[\mathrm{FO}] \text{ then } \mathbf{L}^{\mathbf{NP}} = \mathbf{PSPACE}.$$

Ideally, we would like to equate the statements '$\mathbf{NP} = \mathbf{PSPACE}$' and '$\Omega_b^*[\mathrm{FO}] = \Omega_a^*[\mathrm{FO}]$'; and the statements '$\mathbf{L}^{\mathbf{NP}} = \mathbf{PSPACE}$' and '$(\pm\Omega_b)^*[\mathrm{FO}] = (\pm\Omega_a)^*[\mathrm{FO}]$', but as yet are unable to do so.

Many of our difficulties stem from the fact that we are dealing with the complexity class $\mathbf{NP}$ which is not known to be closed under complementation. If we were considering $\mathbf{P}$, for example, in place of $\mathbf{NP}$ then such difficulties would not arise. However, our results are possible because we can guess a successor relation in NPSB(1) and NPSA(1) (even though the domain of the guessed successor relation might not be the whole of the domain of the input structure), and we do not as yet have a polynomial-time class of program schemes within which we can do likewise. The best polynomial-time class of program schemes we have at present is the class of program schemes NPSS(1), considered in [3], which are essentially the program schemes of NPSA(1) without arrays but with access to a stack (non-determinism is retained, though). It is not clear how we might guess a successor relation within this formalism.

References

[1] S. Abiteboul and V. Vianu, Generic computation and its complexity, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, ACM Press (1991) 209–219.

[2] 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.

[3] 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.

[4] A. Dawar, A restricted second-order logic for finite structures, *Information and Computation* **143** (1998) 154–174.

[5] H.D. Ebbinghaus and J. Flum, *Finite Model Theory*, Springer-Verlag (1995).

[6] J. Esparza and M. Nielsen, Decidability issues for Petri nets – a survey, *Journal of Information Processing and Cybernetics* **30** (1994) 143–160.

[7] M. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman (1979).

[8] G. Gottlob, Relativized logspace and generalized quantifiers over finite ordered structures, *Journal of Symbolic Logic* **62** (1997) 545–574.

[9] M. Grohe, Existential least fixed-point logic and its relatives, *Journal of Logic and Computation* **7** (1997) 205–228.

[10] Y. Gurevich, Logic and the challenge of computer science, *Current Trends in Theoretical Computer Science* (ed. E. Börger), Computer Science Press (1988) 1–57.

[11] N. Immerman, Languages that capture complexity classes, *SIAM Journal of Computing* **16** (1987) 760–778.

[12] N. Immerman, *Descriptive Complexity*, Springer-Verlag (1998).

[13] C. Lautemann, T. Schwentick and I.A. Stewart, Positive versions of polynomial time, *Information and Computation* **147** (1998) 145–170

[14] I.A. Stewart, Complete problems involving boolean labelled structures and projection translations, *Journal of Logic and Computation* **1** (1991) 861–882.

[15] I.A. Stewart, Using the Hamiltonian path operator to capture NP, *Journal of Computer and System Sciences* **45** (1992) 127–151.

[16] I.A. Stewart, Logical and schematic characterization of complexity classes, *Acta Informatica* **30** (1993) 61–87.

[17] I.A. Stewart, Logical characterizations of bounded query classes II: polynomial-time oracle machines, *Fundamenta Informaticae* **18** (1993) 93–105.

[18] I.A. Stewart, Logical description of monotone NP problems, *J. Logic Computat.* **4** (1994) 337–357.

[19] I.A. Stewart, Complete problems for monotone NP, *Theoretical Computer Science* **145** (1995) 147–157.

[20] I.A. Stewart, Logics with zero-one laws that are not fragments of bounded-variable infinitary logic, *Mathematical Logic Quarterly* **41** (1997) 158–178.

[21] I.A. Stewart, Program schemes, arrays, Lindström quantifiers and zero-one laws, *Proceedings of Computer Science Logic 1999*, *Lecture Notes in Computer Science Volume* 1683, Springer-Verlag (1999) 374–388.