Abstract State Machines: Verification Problems and Complexity

Dissertation

Marc Spielmann

Rheinisch-Westfälische Technische Hochschule Aachen

June 2000

Abstract

Abstract state machines (ASMs) provide the formal foundation for a successful methodology for specification and verification of complex dynamic systems. In addition, ASMs induce a computation model on structures, which—in some sense—is more powerful and universal than the standard computation models in theoretical computer science. An investigation of ASMs is therefore interesting from both the point of view of applied computer science and the point of view of theoretical computer science. In the present thesis, practically relevant as well as theoretically motivated questions concerning ASMs are investigated. Subject of the first part of the thesis is the *automatic verifiability* of ASM specifications. In the second part, the ASM computation model itself and *choiceless complexity classes*, which have recently been defined by means of ASMs, are discussed.

Acknowledgements

I would like to thank my advisor, Erich Grädel, for many stimulating discussions, helpful suggestions, and providing direction for my research. Without his advice and support this thesis would not have been possible. A special debt of gratitude I owe to Eric Rosen for numerous conversations on related and unrelated topics and for his countless but futile attempts to improve my English. I am grateful to Jan Van den Bussche for volunteering to review the thesis and for becoming a member of my dissertation committee. Thanks are also due to Andreas Blass and Yuri Gurevich for valuable comments and suggestions concerning the 'choiceless' part of this thesis.

Contents

Introduction

Abstract state machines (ASMs, formerly called evolving algebras) [Gur91, Gur95, Gur97b, Gur99] provide the formal foundation for a successful methodology for specification and verification of complex dynamic systems. The design of complex hardware or software systems is typically organized as a series of refinement steps: starting with a high-level description of the system under consideration, one stepwise refines intermediate description stages until a low-level description is obtained. Ideally, the last description stage is close to an actual implementation. The *ASM method* now proposes to describe each stage of the refinement process in terms of ASMs. The advantage of ASMs is that they are close to logic, which makes the overall design easily amenable to well-understood mathematical techniques. Essentially the mathematical foundation of ASMs supports the formal verification of dynamic systems designed by means of ASMs. For a comprehensive introduction to the ASM method the reader is referred to [Bör99].

Although ASMs can be seen merely as a specification formalism, strictly speaking, ASMs constitute a *computation model on structures*. The program of an ASM is—like the program of a Turing machine—a description of how to modify the current configuration of a machine in order to obtain a possible successor configuration. The main difference between an ASM and a Turing machine is that the configurations of an ASM are *mathematical structures* rather than strings. (We view the work tapes of a Turing machine as strings. Notice that strings are particularly simple, one-dimensional structures. Consequently, every Turing machine can be viewed a particularly simple ASM.) ASMs perform computations on structures in the sense that they obtain a structure as input, modify this structure step by step, and output the resulting structure when reaching a halting state. (Aside from this basic model, there exist more general types of ASM, e.g., real-time ASMs and distributed ASMs [BGR95, Gur95, GR00].) The fact that ASMs work on structures rather than strings has important consequences: the ASM computation model is, in some sense, more powerful and universal than the standard computation models in theoretical computer science [Gur91, Gur95, Gur99].

In this thesis, we study ASMs from a practical as well as a theoretical perspective. In the first part, we regard the ASM language as a specification formalism that has been proved useful in practice, and investigate the *automatic verifiability*

of dynamic systems formalized in terms of ASMs. In the second part, we consider the ASM computation model itself and discuss *choiceless complexity classes* defined by means of ASMs.

Part I: Automatic Verification

The success of the ASM method is witnessed by numerous publications using ASMs for rigorous mathematical correctness proofs of large-scale applications [BH98]. Interestingly, most of the contributions that can be found in the literature focus on *manual* verification, while the number of contributions where all or part of the verification process is *automated* is rather small (for exceptions again consult [BH98]). In a nutshell, computer-aided verification of ASMs, i.e., the (semi-) automatic verification of dynamic systems formalized in terms of ASMs, has not yet been well developed. In the first part of the thesis, we investigate the automatic verifiability of ASMs. More precisely, we view the problem of verifying ASMs as a decision problem of the following kind and investigate the decidability of (a yet to be made precise version of) this problem:

> Given an ASM M and a correctness property φ, decide whether M satisfies φ on all inputs.

Since ASMs are computationally complete—they can calculate all computable functions—this problem is, in its full generality, undecidable. (It is an easy exercise to define a reduction of the halting problem for Turing machines to the verification problem for ASMs.) Therefore, we examine the decidability of the verification problem with respect to *classes of restricted ASMs*. That is, we attempt to identify conditions on ASMs such that the verification problem becomes decidable for ASMs satisfying these conditions. Once decidability with respect to a class of restricted ASMs is established, all ASMs in this class are automatically verifiable in the sense that there exists a procedure which decides whether a given ASM satisfies a given correctness property on all inputs.

Outline of Part I. The first part consists of three chapters, each of which can be read independently. In Chapter 1, we formally define the verification problem for ASMs, present a class of restricted ASMs that can be verified automatically, and show that for straightforward extensions of this class the verification problem becomes undecidable. In Chapter 2, we focus our attention on the automatic verifiability of ASMs specially tailored for electronic commerce applications. All positive verifiability results in the first two chapters are based on reductions to logical decision problems, which are proved decidable in Chapter 3. We conclude our investigation of the automatic verifiability of ASMs with a discussion at the end of Part I.

Part II: Choiceless Complexity

The standard computation model in complexity theory is the Turing machine. Of course, there is no problem with this computation model as long as we consider computational problems whose input instances are strings. However, many computational problems arising in computer science and logic have input instances that are naturally viewed as structures rather than strings. For example, consider the problem of evaluating a database query Q, i.e., a mapping from databases to answer relations: given a database $\mathcal{D}$, i.e., a finite collection of relations on some finite domain, compute the answer relation of Q on $\mathcal{D}$. The input instances of this problem are databases, which are structures. At first glance there is no problem with Turing machines for one can always encode structures as strings and in this way make structures amenable to Turing machines. Unfortunately, this approach has a subtle but important disadvantage: all known string representations of structures generally incorporate superfluous data into the encodings of structures, where by superfluous data we mean data that was originally not contained in the structures (such as a linear order on the universe of a structure). The problem now is that this superfluous data can be retrieved and 'misused' by a Turing machine. Here is an example.

Fix some standard encoding of databases and consider the class of polynomial-time bounded Turing machines computing mappings from (string representations of) databases to (string representations of) answer relations. In general, this class contains a Turing machine which, on input of a database $\mathcal{D}$, computes an answer relation that depends on the *string representation* of $\mathcal{D}$, and not necessarily on $\mathcal{D}$ alone. More to the point, one can find a Turing machine that computes on two different string representations of one and the same database two different, in particular, non-isomorphic, answer relations. The mapping computed by this Turing machine is not considered to be a database query for one is usually not interested in queries whose answer relations depend on the internal representation of databases.

The above example raises an interesting question: Does there exist a polynomial-time computable *string representation of isomorphism classes of structures*? (A computable string representation of isomorphism classes of graphs is given by a full-invariant algorithm I: on input of a finite ordered graph $(\mathcal{G}, <)$, I outputs a bit string $I(\mathcal{G}, <)$ such that for all inputs $(\mathcal{G}_1, <_1)$ and $(\mathcal{G}_2, <_2)$, $I(\mathcal{G}_1, <_1) = I(\mathcal{G}_2, <_2)$ iff $\mathcal{G}_1 \cong \mathcal{G}_2$ [Gur97a].) This question is interesting because the existence of such a string representation would answer in the affirmative one of the main open questions in database theory, namely whether there exists a query language in which precisely all polynomial-time computable queries are expressible. The latter problem was first addressed by Chandra and Harel [CH82] and later rephrased by Gurevich [Gur88] as a question of existence of a logic that captures PTIME. Unfortunately, it is not known whether such a string representation exists.

Motivated, on the one hand, by the quest for a logic that captures PTIME and, on the other hand, by Gurevich's conjecture that no such logic exists [Gur88], Blass, Gurevich, and Shelah have recently asked how much of PTIME can be captured and have put forward a new complexity class that can be viewed as the *choiceless fragment of* PTIME [BGS99]. This class, which they called *Choiceless Polynomial Time* and denoted by $\tilde{\text{C}}$PTIME, is defined by means of ASMs. The advantage of ASMs over Turing machines is that they work directly on structures and treat isomorphic structures in the same way. There simply is no encoding problem. It is worth noticing that, aside from ASMs, several other computation models on structures have been proposed in the literature (see [AHV95, BGS99, BGVdB99, Gur99] and the references there). However, it appears that ASMs are more appropriate for an investigation of $\tilde{\text{C}}$PTIME (see the discussions in [BGS99, Section 1] and [BGVdB99, Section 3]).

In the second part of the thesis, we continue the work of Blass, Gurevich, and Shelah in [BGS99] and study the *choiceless fragment of* LOGSPACE. From the ASM model developed in [BGS99] we derive a restricted model specially tailored for describing logarithmic-space computable functions from structures to structures. Our model is interesting for two reasons. Firstly, it naturally leads to the definition of a complexity class that can be regarded as the logarithmic-space counterpart of $\tilde{\text{C}}$PTIME. We show that this class, which we call *Choiceless Logarithmic Space* and denote by $\tilde{\text{C}}$LOGSPACE, is a proper subclass of both LOGSPACE and $\tilde{\text{C}}$PTIME, thereby separating LOGSPACE and PTIME on the choiceless level. The following figure illustrates our results concerning $\tilde{\text{C}}$LOGSPACE in the context of previous work of Grädel and McColm [GM95] and Blass, Gurevich, and Shelah [BGS99]. It shows the relations between various standard, choiceless, and descriptive complexity classes:

$$\begin{array}{ccccc}
\text{LOGSPACE} & \subseteq & \text{PTIME} & & \text{(standard)} \\
\cup & & \cup & & \\
\tilde{\text{C}}\text{LOGSPACE} & \subset & \tilde{\text{C}}\text{PTIME} & & \text{(choiceless)} \\
\cup & & \cup & & \\
\text{(FO+DTC)} & \subset & \text{(FO+LFP)} & & \text{(descriptive)}
\end{array}$$

where (FO+DTC) and (FO+LFP) denote the classes of problems expressible in deterministic transitive-closure logic and least fixed-point logic, respectively. Notice that all but the uppermost inclusion in this figure are proper inclusions. The question whether LOGSPACE and PTIME are separate classes is a main open problem in complexity theory.

Secondly, our model can serve as a basis for a *reduction theory among structures*. Logarithmic-space computable reductions are widely accepted as a natural basis for completeness results for important (string) complexity classes like PTIME, NPTIME, and PSPACE. Indeed, most of the complete problems for these classes are complete with respect to logarithmic-space computable reductions (see, e.g., [Pap94, GHR95]). Since all ASMs of our restricted model run in logarithmic space, they can serve as a convenient vehicle for describing reductions among structures (that are reductions between computational problems whose input instances are structures). The obtained notion of reducibility among structures subsumes the standard notion of logarithmic-space reducibility.

Outline of Part II. The second part consists of two chapters. In Chapter 5, we introduce our restricted ASM model for describing logarithmic-space computable functions on structures and define the complexity class $\tilde{\text{C}}$LOGSPACE. In Chapter 6, we show that suitable extensions of deterministic transitive-closure logic and least fixed-point logic capture $\tilde{\text{C}}$LOGSPACE and (an extension of) $\tilde{\text{C}}$PTIME, respectively. This link between the two choiceless complexity classes and logic enables us to separate LOGSPACE and PTIME on the choiceless level. We close with a discussion of our results in Part II.

Preliminaries

We recall some notation and terminology from logic and provide a short introduction to ASMs. Readers familiar with ASMs may skip this chapter and refer back to it as needed.

Logic

Vocabularies. A *vocabulary* Υ is a set of relation and function symbols. Each symbol in Υ is associated with a natural number, called the *arity* of the symbol. Symbols of arity 0 are frequently referred to as *nullary symbols*. A *boolean* (resp. *constant*) *symbol* is a nullary relation (resp. function) symbol. If not mentioned otherwise, it is tacitly assumed that every vocabulary is finite and contains (at least) the two constant symbols 0 and 1. A *relational vocabulary* is a vocabulary without function symbols of arity > 0. In particular, **relational vocabularies may contain constant symbols**.

Structures. Let Υ be a vocabulary. A *structure* $\mathcal{A}$ over Υ consists of a nonempty set A, an interpretation $R^{\mathcal{A}} \subseteq A^k$ for every k-ary relation symbol $R \in \Upsilon$, and an interpretation $f^{\mathcal{A}} : A^k \to A$ for every k-ary function symbol $f \in \Upsilon$. The set A is also called the *universe* of $\mathcal{A}$. A *finite structure* is a structure whose universe is finite. The class of finite structures over Υ is denoted by $\mathrm{Fin}(\Upsilon)$. We assume that every structure $\mathcal{A}$ over Υ satisfies the following conditions:

- If Υ contains the constant symbols 0 and 1, then $0^{\mathcal{A}} \neq 1^{\mathcal{A}}$.
- If Υ contains the binary relation symbol $<$, then $<^{\mathcal{A}}$ is a linear order on A, in which case $\mathcal{A}$ is called an *ordered structure*.
- If Υ contains the unary function symbol *succ* and the constant symbol 0, then $succ^{\mathcal{A}}$ is the successor function induced by some linear order on A, and $0^{\mathcal{A}}$ is the least element with respect to this order. In that case, $\mathcal{A}$ is called a *successor structure*.

For every $\Upsilon' \subseteq \Upsilon$, $\mathcal{A}|\Upsilon'$ denotes the *reduct* of $\mathcal{A}$ to Υ', i.e., the structure over Υ' obtained from $\mathcal{A}$ by removing the interpretations of the symbols in $\Upsilon - \Upsilon'$.

First-Order Logic. By FO we denote *first-order logic* with equality. QF is the *quantifier-free fragment* of FO. We freely use the notion of a *logic* (see, e.g., [EFT94]). For a logic L and a formula φ we write $\varphi \in L$ to indicate that φ is a formula of the logic L. The set of formulas $\varphi \in L$ over a particular vocabulary Υ is denoted by $L(\Upsilon)$.

For every $\varphi \in \mathrm{FO}$, $\mathrm{free}(\varphi)$ denotes the set of *free variables* of φ. Sometimes we write $\varphi(x_1, \ldots, x_k)$ to indicate that the variables $x_1, \ldots, x_k$ are pairwise distinct and may occur free in φ, without implying that $\{x_1, \ldots, x_k\} \subseteq \mathrm{free}(\varphi)$ or $\mathrm{free}(\varphi) \subseteq \{x_1, \ldots, x_k\}$. If t and t' are two terms, then $\varphi[t/t']$ stands for the formula obtained from φ by replacing every occurrence of t in φ with t'. The notation $\varphi[./.]$ is also used to denote substitutions of formulas.

Queries. Let k be a natural number and let C be a subclass of $\mathrm{Fin}(\Upsilon)$ closed under isomorphisms (i.e., whenever $\mathcal{A} \in C$ is isomorphic to some $\mathcal{A}' \in \mathrm{Fin}(\Upsilon)$, then $\mathcal{A}' \in C$). A k-ary *query* Q on C is a function that maps every $\mathcal{A} \in C$ to a k-ary relation $Q^{\mathcal{A}} \subseteq A^k$ such that the following condition is satisfied: every isomorphism from a structure $\mathcal{A} \in C$ to a structure $\mathcal{A}'$ is also an isomorphism from $(A, Q^{\mathcal{A}})$ to $(A', Q^{\mathcal{A}'})$. In the special case $k = 0$, Q is also called a *boolean query*. Often, we identify a boolean query Q on $\mathrm{Fin}(\Upsilon)$ with the class of those $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ for which $Q^{\mathcal{A}}$ is true.

Note that every $\varphi(x_1, \ldots, x_k) \in \mathrm{FO}(\Upsilon)$ with $\mathrm{free}(\varphi) \subseteq \{x_1, \ldots, x_k\}$ defines a k-ary query on $\mathrm{Fin}(\Upsilon)$. This query maps any $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ to the k-ary relation

$$\varphi^{\mathcal{A}} \;:=\; \{(a_1, \ldots, a_k) \in A^k : \mathcal{A} \models \varphi[a_1, \ldots, a_k]\},$$

where the notation $\mathcal{A} \models \varphi[a_1, \ldots, a_k]$ indicates that φ holds in the structure obtained from $\mathcal{A}$ by adding the elements $a_1, \ldots, a_k$ as interpretations of the variables $x_1, \ldots, x_k$.

Transitive-Closure Logic. *Transitive-closure logic*, denoted (FO+TC), is obtained by adding to first-order logic a transitive-closure operator (TC), which assigns to any definable $2k$-ary relation R the transitive, reflexive closure of R. Formally, (FO+TC) is obtained from FO by means of the following additional formula-formation rule:

(TC) If φ is a formula, $\bar{x}$ and $\bar{x}'$ are two k-tuples of variables such that the variables among $\bar{x}, \bar{x}'$ are pairwise distinct, and $\bar{t}$ and $\bar{t}'$ are two k-tuples of terms, then $[\mathrm{TC}_{\bar{x},\bar{x}'}\, \varphi](\bar{t}, \bar{t}')$ is a formula.

A formula of the form $[\mathrm{TC}_{\bar{x},\bar{x}'}\, \varphi](\bar{t}, \bar{t}')$ is also referred to as a TC *formula*. A variable occurs free in a TC formula $[\mathrm{TC}_{\bar{x},\bar{x}'}\, \varphi](\bar{t}, \bar{t}')$ if it occurs in $\bar{t}$ or $\bar{t}'$, or if it occurs free in φ and is different from any variable among $\bar{x}, \bar{x}'$.

Intuitively, the meaning of a TC formula $[\mathrm{TC}_{\bar{x},\bar{x}'}\, \varphi](\bar{t}, \bar{t}')$ in the context of a finite structure $\mathcal{A}$ is as follows. Regard $[\mathrm{TC}_{\bar{x},\bar{x}'}\, \varphi]$ as a new $2k$-ary relation symbol whose interpretation is the transitive, reflexive closure of the image of $\mathcal{A}$ under

the $2k$-ary query defined by $\varphi(\bar{x}, \bar{x}')$. For example, consider a directed graph $\mathcal{G} = (V, E)$ with node set V and binary edge relation E. For any two nodes $a, b \in V$, $\mathcal{G} \models [\mathrm{TC}_{x,y}\, E(x,y) \vee E(y,x)][a,b]$ iff there is an undirected path in $\mathcal{G}$ connecting a and b. For a formal definition of the semantics of TC formulas see, e.g., [EF95].

(E+TC) denotes *existential transitive-closure logic*, i.e., the existential fragment of (FO+TC). The formulas of (E+TC) are built from atomic and negated atomic formulas by means of disjunction, conjunction, existential quantification, and the TC operator.

Deterministic Transitive-Closure Logic. *Deterministic transitive-closure logic*, denoted (FO+DTC), is first-order logic augmented with a deterministic transitive-closure operator (DTC), which assigns to any definable $2k$-ary relation R the transitive, reflexive closure of the deterministic part of R. (The deterministic part of a binary relation E, e.g., is obtained from E by removing all edges that start at a node with out-degree ≥ 2.) Formally, (FO+DTC) is defined as (FO+TC) above, except that now rule **(TC)** is replaced with the following rule:

(DTC) If φ is a formula, $\bar{x}$ and $\bar{x}'$ are two k-tuples of variables such that the variables among $\bar{x}, \bar{x}'$ are pairwise distinct, and $\bar{t}$ and $\bar{t}'$ are two k-tuples of terms, then $[\mathrm{DTC}_{\bar{x},\bar{x}'}\, \varphi](\bar{t}, \bar{t}')$ is a formula.

The *free* and *bound variables* of (FO+DTC) formulas are defined as for (FO+TC) formulas.

The semantics of a DTC formulas $[\mathrm{DTC}_{\bar{x},\bar{x}'}\, \varphi](\bar{t}, \bar{t}')$ is given by the semantics of the TC formula $[\mathrm{TC}_{\bar{x},\bar{x}'}\, \varphi \wedge \forall \bar{y}(\varphi[\bar{x}'/\bar{y}] \rightarrow \bar{y} = \bar{x}')](\bar{t}, \bar{t}')$, presuming that none of variables in $\bar{y}$ occurs in φ. For instance, if $\mathcal{G} = (V, E)$ is a directed graph and $a, b \in V$ are two nodes in $\mathcal{G}$, then $\mathcal{G} \models [\mathrm{DTC}_{x,y}\, E(x,y)][a,b]$ iff there exists a directed path in $\mathcal{G}$ which starts at node a, ends at node b, and each node on that path, except b, has out-degree 1.

Finite Satisfiability and Finite Validity. Let L be a logic and let Υ a vocabulary. Consider a sentence $\varphi \in L(\Upsilon)$, i.e., a formula in $L(\Upsilon)$ without free variables. φ is *finitely satisfiable* if there exists $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ with $\mathcal{A} \models \varphi$. φ is *finitely valid* if for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, $\mathcal{A} \models \varphi$.

By FIN-SAT(L) (resp. FIN-VAL(L)) we denote the problem of deciding finite satisfiability (resp. finite validity) of a given sentence $\varphi \in L$. For every natural number m, FIN-SAT$_m(L)$ and FIN-VAL$_m(L)$ denote the restrictions of FIN-SAT(L) and FIN-VAL(L), respectively, to sentences in which only symbols of arity at most m occur.

Abstract State Machines

ASM Vocabularies. An *ASM vocabulary* Υ is a quadruple (Υ_{in}, Υ_{stat}, Υ_{dyn}, Υ_{out}) of pairwise disjoint vocabularies where (only) Υ_{in} contains the two constant

symbols 0 and 1. The symbols in Υ_{in}, Υ_{stat}, Υ_{dyn}, and Υ_{out} are called *input*, *static*, *dynamic*, and *output symbols*, respectively. Sometimes we also denote by Υ the (ordinary) vocabulary $\Upsilon_{\text{in}} \cup \Upsilon_{\text{stat}} \cup \Upsilon_{\text{dyn}} \cup \Upsilon_{\text{out}}$. The intended meaning will be clear from the context.

Remark 1. Generally, ASM vocabularies also contain *external symbols*, that are symbols whose interpretation is exclusively determined by the environment of an ASM. ASMs with external relations or functions can be viewed as interactive machines and will be considered in Chapter 2 (see Remark 2.1.4 (b)). □

States. Let Υ be an ASM vocabulary. A *state* $\mathcal{S}$ over Υ is a structure over the (ordinary) vocabulary Υ. We consider both finite and infinite states of ASMs. To ease notation, we frequently write $\mathcal{S}|_{\text{in}}$ instead of $\mathcal{S}|\Upsilon_{\text{in}}$ (i.e., the reduct of $\mathcal{S}$ to Υ_{in}), $\mathcal{S}|_{\text{in,dyn}}$ instead of $\mathcal{S}|(\Upsilon_{\text{in}} \cup \Upsilon_{\text{dyn}})$, and so forth. Often it is convenient to view a state $\mathcal{S}$ as being composed of the four components $\mathcal{S}|_{\text{in}}$, $\mathcal{S}|_{\text{stat}}$, $\mathcal{S}|_{\text{dyn}}$, and $\mathcal{S}|_{\text{out}}$, which we call the *input*, *static*, *dynamic*, and *output component* of $\mathcal{S}$, respectively. The meaning of each component is as follows:

- $\mathcal{S}|_{\text{in}}$ is the *input* of an ASM in state $\mathcal{S}$.
- $\mathcal{S}|_{\text{stat}}$ stores all *static data* that is not included in the input, such as arithmetical operations, etc. Both $\mathcal{S}|_{\text{in}}$ and $\mathcal{S}|_{\text{stat}}$ do not change during a computation.
- $\mathcal{S}|_{\text{dyn}}$ contains all *dynamic data*, such as program variables, arrays, etc. It may change in every computation step.
- $\mathcal{S}|_{\text{out}}$ is the current *output*.

ASM Programs. Fix an ASM vocabulary Υ. A *guard* φ is a FO formula over $\Upsilon - \Upsilon_{\text{out}}$. *ASM programs* are defined inductively:

- **Updates:** For every relation symbol $R \in \Upsilon_{\text{dyn}} \cup \Upsilon_{\text{out}}$, every function symbol $f \in \Upsilon_{\text{dyn}} \cup \Upsilon_{\text{out}}$, and all terms $\bar{t}, t_0$ over $\Upsilon - \Upsilon_{\text{out}}$, each of the following is a program: $R(\bar{t})$, $\neg R(\bar{t})$, $f(\bar{t}) := t_0$. Such programs are also called *atomic programs*.
- **Conditionals:** If Π is a program and φ is a guard, then

  `if` φ `then`
  Π

  is a program.

- **Parallel composition:** If Π_0 and Π_1 are programs, then

  Π_0
  Π_1

  is a program. Sometimes this program is also denoted by $(\Pi_0||\Pi_1)$.

- **Parameterized parallel composition:** If Π is a program, $\bar{x}$ is a tuple of pairwise distinct variables, and φ is a guard, then

  `do-for-all` $\bar{x} : \varphi$
  Π

  is a program.

- **Non-deterministic choice:** If Π is a program, $\bar{x}$ is a tuple of pairwise distinct variables, and φ is a guard, then

  `choose` $\bar{x} : \varphi$
  Π

  is a program.

The *guards* and the *free* and *bound variables* of an ASM program are defined in the obvious way. Generally, we are only interested in ASM programs without free variables. Thus, if not stated otherwise, it is tacitly assumed that every ASM program occurring in the text has no free variables. (Notice that one can always eliminate free variables by replacing them with new constant symbols.) An ASM program is called *deterministic* if it does not contain `choose`.

Remark 2. (a) Often, ASM programs also contain import rules which enable ASMs to invent new objects [Gur97b, Gur95]. In Part II, we will adopt from [BGS99] a very elegant method for object invention without import rules (see Section 5.1).

(b) In view of the fact that the states of an ASM can be infinite, one may question our decision to allow unbounded first-order quantification in the guards of ASM programs. We address this issue in Remark 3 below. □

Transitions. Consider an ASM program Π. Π can be viewed as a description of how to modify the current state of an ASM in order to obtain a possible successor state. Formally, Π defines a transition relation $Trans_\Pi$ on states as follows.

An *update set* U over some ASM vocabulary Υ is a set of atomic programs over Υ. Let $\mathcal{S}$ and $\mathcal{S}'$ be two states and let U be an update set, each over Υ. $\mathcal{S}'$ is called the *successor state* of $\mathcal{S}$ with respect to U if $\mathcal{S}'$ is identical to $\mathcal{S}$, except of the following modifications:

- if $R(\bar{t}) \in U$ and there is no update $\neg R(\bar{s}) \in U$ such that $\mathcal{S} \models (\bar{t} = \bar{s})$, then $\mathcal{S}' \models R(\bar{t})$,

- if $\neg R(\bar{t}) \in U$ and there is no update $R(\bar{s}) \in U$ such that $\mathcal{S} \models (\bar{t} = \bar{s})$, then $\mathcal{S}' \models \neg R(\bar{t})$,
- if $(f(\bar{t}) := t_0) \in U$ and there is no update $(f(\bar{s}) := s_0) \in U$ such that $\mathcal{S} \models (\bar{t} = \bar{s}) \wedge (t_0 \neq s_0)$, then $\mathcal{S}' \models (f(\bar{t}) = t_0)$.

Informally, $\mathcal{S}'$ is obtained from $\mathcal{S}$ and U by modifying $\mathcal{S}$ so that every atomic and negated atomic formula corresponding to a consistent update in U is satisfied in $\mathcal{S}'$.

We say that a state $\mathcal{S}$ is *appropriate* for Π if Π is an ASM program over the vocabulary of $\mathcal{S}$. Simultaneously for all states $\mathcal{S}$ appropriate for Π, define a set $\mathrm{Den}(\Pi, \mathcal{S})$ of update sets by induction on the construction of Π. If Π is an atomic program, then let $\mathrm{Den}(\Pi, \mathcal{S}) := \big\{\{\Pi\}\big\}$. If $\Pi = (\texttt{if } \varphi \texttt{ then } \Pi_0)$, then let

$$\mathrm{Den}(\Pi, \mathcal{S}) \;:=\; \begin{cases} \mathrm{Den}(\Pi_0, \mathcal{S}) & \text{if } \mathcal{S} \models \varphi \\ \{\varnothing\} & \text{otherwise.} \end{cases}$$

If $\Pi = (\Pi_0 || \Pi_1)$, then let

$$\mathrm{Den}(\Pi, \mathcal{S}) \;:=\; \big\{U_0 \cup U_1 : U_0 \in \mathrm{Den}(\Pi_0, \mathcal{S}), U_1 \in \mathrm{Den}(\Pi_1, \mathcal{S})\big\}.$$

Suppose that $\Pi = \big(\texttt{do-for-all } \bar{x} : \varphi(\bar{x}),\, \Pi_0\big)$. Choose an index set I such that there exists a one-to-one mapping m from I to $\{\bar{a} : \mathcal{S} \models \varphi[\bar{a}]\}$. For each $i \in I$, set $\bar{a}_i = m(i)$. Let

$$\mathrm{Den}(\Pi, \mathcal{S}) \;:=\; \Big\{\bigcup\nolimits_{i \in I} U_i : U_i \in \mathrm{Den}(\Pi_0, (\mathcal{S}, \bar{a}_i))\Big\},$$

where $(\mathcal{S}, \bar{a}_i)$ denotes the state obtained from $\mathcal{S}$ by adding the j-th element in $\bar{a}_i$ as an interpretation of the j-th variable in $\bar{x}$. (Observe that some of the variables in $\bar{x}$ may occur free in Π_0. If we view all free occurrences of these variables as new constant symbols, then $\mathrm{Den}(\Pi_0, \mathcal{S}')$ is defined for states $\mathcal{S}'$ which include interpretations of the new symbols.) Finally, if $\Pi = (\texttt{choose } \bar{x} : \varphi(\bar{x}),\, \Pi_0)$, then let

$$\mathrm{Den}(\Pi, \mathcal{S}) \;:=\; \begin{cases} \bigcup_{\bar{a} \in \varphi^{\mathcal{S}}} \mathrm{Den}(\Pi_0, (\mathcal{S}, \bar{a})) & \text{if } \varphi^{\mathcal{S}} \neq \varnothing \\ \{\varnothing\} & \text{otherwise.} \end{cases}$$

Notice that, if Π is deterministic, then $\mathrm{Den}(\Pi, \mathcal{S})$ is a singleton set.

The *transition relation* $Trans_\Pi$ (defined by Π) is a binary relation between states. It contains a pair $(\mathcal{S}, \mathcal{S}')$ of states iff

1. both $\mathcal{S}$ and $\mathcal{S}'$ are appropriate for Π, and
2. there exists an update set $U \in \mathrm{Den}(\Pi, \mathcal{S})$ such that $\mathcal{S}'$ is the successor state of $\mathcal{S}$ with respect to U.

If $(\mathcal{S}, \mathcal{S}') \in \mathit{Trans}_\Pi$, then $\mathcal{S}'$ is also called a *successor state* of $\mathcal{S}$ with respect to Π. It is easy to see that, if Π is deterministic, then Trans_Π is deterministic, i.e., whenever $(\mathcal{S}_1, \mathcal{S}_1')$ and $(\mathcal{S}_2, \mathcal{S}_2')$ are in Trans_Π and $\mathcal{S}_1 = \mathcal{S}_2$, then $\mathcal{S}_1' = \mathcal{S}_2'$.

Remark 3. Readers familiar with ASMs may have noticed that the above definition of a successor state (called sequel in [Gur97b]) does not entirely conform to the ASM framework [Gur97b, Gur95]. Our notion of successor state differs from the standard notion in two ways:

1. According to the above definition, the successor state of a state $\mathcal{S}$ with respect to an inconsistent update set can be different from $\mathcal{S}$. In contract, any inconsistent update set is treated as the empty update set in [Gur97b].

   This deviation from the standard notion is not essential for the validity of (most of) our results and allows us to simplify the presentation of technicalities significantly. The only results presented in this thesis that may not hold with respect to the standard semantics are those concerning the verifiability of ASM transducers with input-bounded quantification (see the second part of Section 2.4).

2. In [Gur97b] the range of variables quantified by means of `do-for-all`, `choose`, or first-order quantifiers in guards is implicitly bounded to the set of non-reserve elements. According to our definition, the range of such variables is a priori unbounded.

   Note that in both cases we may encounter situations where $\mathrm{Den}(\Pi, \mathcal{S})$ contains infinitely many update sets or update sets that are infinite. In such cases, we may not be able to actually compute one or all successor states of $\mathcal{S}$ with respect to Π. Here, we assume the point of view that it is in the responsibility of the user of ASMs to ensure finiteness of the occurring update sets when modeling feasible algorithms that operate on infinite states.

□

Equivalence. Two ASM programs are *equivalent* if they define the same transition relation.

Inputs. An *input* over an ASM vocabulary Υ is a finite structure over Υ_{in}.

Initialization Mappings. Let Υ be an ASM vocabulary. An *initialization mapping* over Υ is a function that maps every input $\mathcal{I}$ over Υ to a state $\mathcal{S}_\mathcal{I}$ over Υ satisfying the following conditions:

1. The universe of $\mathcal{I}$ is a subset of the universe of $\mathcal{S}_\mathcal{I}$.

2. For every relation symbol $R \in \Upsilon_{\mathrm{in}}$, every k-ary function symbol $f \in \Upsilon_{\mathrm{in}}$, and every k-tuple $\bar{a}$ of elements of $\mathcal{S}_\mathcal{I}$

$$R^{\mathcal{S}_\mathcal{I}} = R^\mathcal{I} \quad \text{and} \quad f^{\mathcal{S}_\mathcal{I}}(\bar{a}) = \begin{cases} f^\mathcal{I}(\bar{a}) & \text{if } \bar{a} \text{ consists of elements of } \mathcal{I} \\ 0^\mathcal{I} & \text{otherwise.} \end{cases}$$

3. For every input $\mathcal{I}'$ over Υ, if $\mathcal{I}'$ and $\mathcal{I}$ are isomorphic, so are $\mathcal{S}_{\mathcal{I}'}$ and $\mathcal{S}_{\mathcal{I}}$.

Abstract State Machines. An *abstract state machine* M is a triple $(\Upsilon, \mathit{initial}, \Pi)$ consisting of an ASM vocabulary Υ, an initialization mapping $\mathit{initial}$ over Υ, and an ASM program Π over Υ. We call Υ_{in}, Υ_{stat}, Υ_{dyn}, and Υ_{out} the *input*, *static*, *dynamic*, and *output vocabulary* of M, respectively. An *input* appropriate for M is an input over Υ. M is *deterministic* if Π is so.

Runs. Let $M = (\Upsilon, \mathit{initial}, \Pi)$ be an ASM and let $\mathcal{I}$ be an input appropriate for M. A *run* ρ of M on $\mathcal{I}$ is an infinite sequence $(\mathcal{S}_i)_{i\in\omega}$ of states over Υ such that $\mathcal{S}_0 = \mathit{initial}(\mathcal{I})$ and for every $i \in \omega$, $(\mathcal{S}_i, \mathcal{S}_{i+1}) \in \mathit{Trans}_\Pi$.

Outputs. There exist various meaningful definitions of the output produced by an ASM during a run. Which one is more appropriate depends on the context in which ASMs are employed. For that reason, we do not fix a specific convention at this point and instead provide the corresponding definitions in the text (see Sections 2.1 and 5.2).

I

Automatic Verification

1

Verification of Abstract State Machines

Consider a programming language L and a formal language F suitable to express properties of programs in L. The problem of verifying L-programs against F-properties can be seen as a decision problem of the following kind:

> Given a program $\Pi \in L$ and a property $\varphi \in F$, decide whether for *every* input $\mathcal{I}$ appropriate for Π, Π on input $\mathcal{I}$ satisfies φ.

Decidability of this problem obviously depends on the expressiveness of both L and F. For example, we will not be able to decide the problem for Turing machine programs and the fixed property "a halting configuration is reachable" since the resulting instance of the problem coincides with the halting problem for Turing machines.

In this chapter, we investigate the decidability of the above problem for the ASM 'programming' language. As our main positive result we present a class of restricted ASMs (that can be viewed as a simple programming language) and a temporal logic (suitable to express properties of ASMs) for which the above problem is decidable. Consequently, our restricted ASMs are automatically verifiable in the following sense: there exists a procedure that decides whether a given restricted ASM satisfies a given property on all inputs. Due to the syntactic restrictions we impose on their programs, we call our restricted ASMs *sequential nullary ASMs* (for details see Section 1.2).

Although the computational power of sequential nullary ASMs is limited, they can still be regarded as software; for the following two reasons. Firstly, the size of the input of a sequential nullary ASM is not a priori bounded. For instance, the input can be any finite graph. Secondly, the course and the result of a

computation of a sequential nullary ASM may depend on a 'non-trivial' portion of the input. Consider, for example, the following decision problem also known as the reachability problem: given a finite graph $\mathcal{G}$ containing two distinguished nodes *source* and *target*, decide whether there exists a path from *source* to *target* in $\mathcal{G}$. Any algorithm that solves the reachability problem for all finite graphs has to explore, in the worst case, the entire input graph in order to compute the right answer. We consider the reachability problem a typical software problem and show that it can be solved by means of a sequential nullary ASM (see Example 1.1.4).

We conclude our investigation of the automatic verifiability of ASMs in this chapter by providing some evidence that for ASMs more powerful than sequential nullary ASMs the verification problem becomes undecidable. More precisely, we show that, if the computational power of sequential nullary ASMs is increased in a straightforward manner, then most basic safety and liveness properties of the resulting ASMs can not be verified automatically.

Related Work. Sequential nullary ASMs share some similarities with the inductive programs studied by Harel and Kozen in [HK84]. In particular, every inductive program of Harel and Kozen that does not employ universal choice can easily be converted into a sequential nullary ASM. As a formalism suitable to express properties of ASMs we propose first-order branching temporal logic (see, e.g., [Eme90]). This logic is a straightforward extension of the well-known propositional branching-time logic CTL* of Clarke, Emerson, and Sistla [CES86] by first-order reasoning. Our main positive result is implied by a reduction to the finite satisfiability problem for existential transitive closure logic, which is decidable by results in [LMSS93, Ros99] (see also Chapter 3). The reduction is inspired by a construction due to Immerman and Vardi that was first presented in [IV97] as a translation from CTL* into transitive closure logic.

Outline of the Chapter. In Section 1.1, we formally define the verification problem for ASMs. Solving this problem coincides with proving properties of ASMs. In Section 1.2, we introduce sequential nullary ASMs and compare their computational power with the computational power of Turing machines. Our results concerning the automatic verifiability of ASMs (i.e., the decidability of verification problem for ASMs) are presented in Section 1.3 and discussed at the end of Part I.

1.1 The Verification Problem for ASMs

Our main concern in this section is to define (an instance of) the following problem in terms of a computational problem:

> Given an ASM M and a correctness property φ, decide whether for every input $\mathcal{I}$ appropriate for M, M on input $\mathcal{I}$ satisfies φ.

The obtained computational problem will be called the *verification problem for ASMs*.

Remark 1.1.1. In applications it often suffices to ensure correctness of an ASMs only for inputs that satisfy certain conditions. Since in the verification problem for ASMs $\mathcal{I}$ varies over *all* inputs appropriate for a given ASM, one may object that this problem does not adequately reflect real-life verification of ASMs. Suppose, for example, that the formula ψ describes precisely the 'nice' inputs of an ASM M, i.e., for every input $\mathcal{I}$ appropriate for M, $\mathcal{I}$ is nice iff $\mathcal{I} \models \psi$. The question is now how to verify whether M satisfies φ on every nice input rather than on every input (appropriate for M)? We will define the verification problem so that M satisfies φ on every nice input iff $(M, \psi \rightarrow \varphi)$ is a positive instance of the verification problem. □

The above informal formulation of the verification problem for ASMs immediately raises the question of a *specification language for ASMs*, i.e., a formal language suitable to express properties of ASMs. Since in the literature there is no consensus on the choice of such a language, we advocate here *first-order branching temporal logic*.

First-Order Branching Temporal Logic

Consider an ASM $M = (\Upsilon, \mathit{initial}, \Pi)$. The runs of M on a particular input can be arranged as a graph as follows:

Definition 1.1.2. Let M be as above and let $\mathcal{I}$ be an input appropriate for M. The *computation graph* of M on $\mathcal{I}$, denoted $C_M(\mathcal{I})$, is a triple $(\mathit{States}, \mathit{Trans}, \mathcal{S}_0)$, where

- *States* is the set of those states over Υ whose universe is identical with the universe of $\mathit{initial}(\mathcal{I})$,
- $\mathit{Trans} \subseteq \mathit{States} \times \mathit{States}$ is the restriction of Trans_Π to *States*, and
- $\mathcal{S}_0 = \mathit{initial}(\mathcal{I})$.

□

Obviously, the infinite paths in $C_M(\mathcal{I})$ starting at $\mathcal{S}_0$ are precisely the runs of M on $\mathcal{I}$. It is thus reasonable to express a property of M (on any input) as a property of *all* computation graphs of M. To express properties of computation graphs we propose first-order branching temporal logic (FBTL), a straightforward extension of the well-known propositional branching-time logic CTL* by first-order reasoning [CES86, Eme90]. On the one hand, first-order logic allows us to reason about the states of an ASM as each such state is a first-order structure.

On the other hand, CTL* enables us to reason about the temporal and nondeterministic behavior of an ASM. For instance, it is possible to express that 'good things eventually happen' or that 'there exists a good run' of an ASM (see the definition of the semantics of FBTL formulas below).

Definition 1.1.3. *State formulas* and *path formulas* of *first-order branching temporal logic* are defined by simultaneous induction:

(S1) Every atomic FO formula is a state formula.

(S2) If φ is a path formula, then $\mathbf{E}\varphi$ is a state formula.

(P1) Every state formula is a path formula.

(P2) If φ and ψ are path formulas, then $\mathbf{X}\varphi$, $\varphi\mathbf{U}\psi$, and $\varphi\mathbf{B}\psi$ are path formulas.

(SP1) If φ and ψ are state (resp. path) formulas, then $\varphi \vee \psi$ and $\neg\varphi$ are state (resp. path) formulas.

(SP2) If x is a variable and φ a state (resp. path) formula, then $\exists x\varphi$ is a state (resp. path) formula.

The *free* and *bound variables* of state and path formulas are defined in the obvious way. FBTL denotes the set of state formulas. □

Semantics of FBTL formulas. Intuitively, a state formula of the form $\mathbf{E}\varphi$ expresses that there (E)xists an infinite path (presumably a run of an ASM) such that the path formula φ holds along this path. The intuitive meaning of path formulas of the form $\mathbf{X}\varphi$, $\varphi\mathbf{U}\psi$, and $\varphi\mathbf{B}\psi$ is as follows:

$\mathbf{X}\varphi$: φ holds in the ne(X)t state.

$\varphi\mathbf{U}\psi$: ψ holds eventually and φ holds (U)ntil then.

$\varphi\mathbf{B}\psi$: either ψ holds always or φ holds (B)efore ψ fails.

We define the semantics of FBTL formulas only with respect to computation graphs of ASMs. Let $C = (\mathit{States}, \mathit{Trans}, \mathcal{S}_0)$ be a computation graph of an ASM of vocabulary Υ, and let $\rho = (\mathcal{S}'_i)_{i\in\omega}$ be an infinite path in C, not necessarily starting at the initial state $\mathcal{S}_0$. For any $j \in \omega$, we denote by $\rho|j$ the infinite path $(\mathcal{S}'_{i+j})_{i\in\omega}$, i.e., the suffix $\mathcal{S}'_j, \mathcal{S}'_{j+1}, \ldots$ of ρ. Let φ (resp. ψ) be a state (resp. path) formula over Υ with $\mathrm{free}(\varphi) = \mathrm{free}(\psi) = \{\bar{x}\}$. Simultaneously for every state $\mathcal{S}$ in C, every infinite paths ρ in C, and all interpretations $\bar{a}$ of the variables $\bar{x}$ (chosen

from the universe of $\mathcal{S}_0$), define the two satisfactory relations $(C, \mathcal{S}, \bar{a}) \models \varphi$ and $(C, \rho, \bar{a}) \models \psi$ by induction on the construction of φ and ψ:

(S1)	$(C, \mathcal{S}, \bar{a}) \models \varphi$	$:\Leftrightarrow$	$\mathcal{S} \models \varphi[\bar{a}]$, if φ is an atomic formula
(S2)	$(C, \mathcal{S}, \bar{a}) \models \mathbf{E}\varphi$	$:\Leftrightarrow$	there is an infinite path ρ' in C starting at $\mathcal{S}$ such that $(C, \rho', \bar{a}) \models \varphi$
(P1)	$(C, \rho, \bar{a}) \models \varphi$	$:\Leftrightarrow$	$(C, \mathcal{S}_0', \bar{a}) \models \varphi$, where $\mathcal{S}_0'$ is the first state in ρ
(P2)	$(C, \rho, \bar{a}) \models \mathbf{X}\varphi$	$:\Leftrightarrow$	$(C, \rho\|1, \bar{a}) \models \varphi$
	$(C, \rho, \bar{a}) \models \varphi\mathbf{U}\psi$	$:\Leftrightarrow$	there is an $i \in \omega$, such that $(C, \rho\|i, \bar{a}) \models \psi$ and for all $j < i$, $(C, \rho\|j, \bar{a}) \models \varphi$
	$(C, \rho, \bar{a}) \models \varphi\mathbf{B}\psi$	$:\Leftrightarrow$	for every $i \in \omega$, if $(C, \rho\|i, \bar{a}) \models \neg\psi$, then there is a $j < i$ with $(C, \rho\|j, \bar{a}) \models \varphi$

The semantics of formulas derived by means of rule **(SP1)** is standard. It remains to declare the semantics of formulas derived by means of rule **(SP2)**. Below, σ stands for either a state $\mathcal{S}$ or an infinite path ρ in C, depending on whether φ is a state or a path formula.

(SP2)	$(C, \sigma, \bar{a}) \models \exists y\varphi(\bar{x}, y)$	$:\Leftrightarrow$	there is an element b in the universe of $\mathcal{S}_0$ such that $(C, \sigma, \bar{a}, b) \models \varphi(\bar{x}, y)$

For every FBTL sentence φ over Υ, let

$$C \models \varphi \quad :\Leftrightarrow \quad (C, \mathcal{S}_0) \models \varphi.$$

The following abbreviations are customary in CTL* and will be used frequently:

$\mathbf{A}\varphi$	:=	$\neg\mathbf{E}\neg\varphi$	(φ holds along every path)
$\mathbf{F}\varphi$	:=	$\mathit{true}\,\mathbf{U}\varphi$	(φ holds eventually)
$\mathbf{G}\varphi$	:=	$\mathit{false}\,\mathbf{B}\varphi$	(φ holds always)

Note that $\varphi\mathbf{B}\psi \equiv \neg(\neg\varphi\mathbf{U}\neg\psi)$.

Example 1.1.4. Consider the following decision problem, which is also known as the *reachability problem* [Pap94]:

REACH : Given a finite directed graph $\mathcal{G} = (V, E)$ and two nodes s and t in $\mathcal{G}$, decide whether there exists a directed path from source s to target t in $\mathcal{G}$.

We present a non-deterministic ASM $M_{\text{reach}} = (\Upsilon, \mathit{initial}, \Pi)$ that solves the reachability problem. M_{reach} takes instances of REACH as input, i.e., finite structures of the form $(\mathcal{G}, s, t)$, and accepts an input iff the input is a positive instance of REACH.

To the definition of Υ and Π. Let E be a binary relation symbol (which will be interpreted as the edge relation of an input graph of M), let *accept* and *running* be two boolean symbols, and let *pebble* be a nullary function symbols. Define the ASM vocabulary Υ by setting $\Upsilon_{\text{in}} = \{E, 0, 1\}$, $\Upsilon_{\text{dyn}} = \{\mathit{accept}, \mathit{running}, \mathit{pebble}\}$, and $\Upsilon_{\text{stat}} = \Upsilon_{\text{out}} = \varnothing$. The program of M_{reach} is displayed below. For clarity, we write *source* instead of 0, and *target* instead of 1.

`program` Π:

```
if ¬running then
    pebble := source
    running := true

if running then
    if pebble ≠ target then
        choose z : true
            if E(pebble, z) then
                pebble := z
    else
        accept
```

To the definition of *initial*. Observe that every instance $\mathcal{I} = (\mathcal{G}, s, t)$ of REACH is a finite structure over Υ_{in}, where $E^{\mathcal{I}}$ is the edge relation of the input graph $\mathcal{G}$ and $0^{\mathcal{I}} = s$ and $1^{\mathcal{I}} = t$. In particular, $\mathcal{I}$ is an input appropriate for M_{reach}. Let the initialization mapping be defined by $\mathit{initial}(\mathcal{I}) := (\mathcal{I}, \mathit{false}, \mathit{false}, 0^{\mathcal{I}})$, where $(\mathcal{I}, \mathit{false}, \mathit{false}, 0^{\mathcal{I}})$ denotes the state over Υ obtained from $\mathcal{I}$ by adding interpretations of *accept*, *running*, and *pebble*.

Consider the computation graph $C_{M_{\text{reach}}}(\mathcal{I})$ of M_{reach} on input $\mathcal{I} = (\mathcal{G}, s, t)$. Every state in $C_{M_{\text{reach}}}(\mathcal{I})$ is a finite structure over Υ of the form $(\mathcal{I}, b_a, b_r, a_p)$ where the interpretations b_a and b_r of *accept* and *running*, respectively, are boolean values, and the interpretation a_p of *pebble* is a node in $\mathcal{G}$. The initial state in $C_{M_{\text{reach}}}(\mathcal{I})$ is $(\mathcal{I}, \mathit{false}, \mathit{false}, 0^{\mathcal{I}})$.

We demonstrate that correctness of M_{reach} can be expressed in terms of FBTL. To be more accurate, we provide a FBTL formula φ over Υ such that φ holds in all computation graphs of M_{reach} iff M_{reach} accepts precisely the positive instances of REACH. Consider the following path formula ψ expressing that, if *pebble* is moved during a computation step of M_{reach}, then it is moved along an edge of the input graph.

$$\psi \;:=\; \forall x\big[x = \mathit{pebble} \rightarrow \mathbf{X}\big(E(x, \mathit{pebble}) \vee x = \mathit{pebble}\big)\big]$$

It is not hard to see that $(C, \rho) \models \mathbf{G}\psi$ for every computation graph C of M_{reach} and every path ρ in C, and that the following equivalences hold for every input $\mathcal{I} = (\mathcal{G}, s, t)$ appropriate for M_{reach}:

$$\begin{aligned} \mathcal{I} \in \text{REACH} \;&\Leftrightarrow\; \mathcal{G} \models [\text{TC}_{x,x'}\, E(x, x')][s, t] \\ &\Leftrightarrow\; C_{M_{\text{reach}}}(\mathcal{I}) \models \mathbf{E}\big(\mathit{pebble} = \mathit{source} \wedge \mathbf{G}\psi \wedge \mathbf{F}(\mathit{pebble} = \mathit{target})\big) \end{aligned}$$

Thus

$$\varphi \;:=\; \mathbf{EF}\mathit{accept} \leftrightarrow \mathbf{E}\big(\mathit{pebble} = \mathit{source} \wedge \mathbf{G}\psi \wedge \mathbf{F}(\mathit{pebble} = \mathit{target})\big)$$

is a FBTL formula expressing correctness of M_{reach}. □

Notice that in the last example we reduced the problem of verifying M_{reach} to the problem of deciding validity of φ with respect to the computation graphs of M_{reach}. Indeed, we next define the verification problem for ASMs as the problem of deciding validity of a given FBTL formula with respect to the computation graphs of a given ASM.

The Verification Problem (revisited)

Recall the informal formulation of the verification problem for ASMs at the beginning of this section. If the reader agrees that FBTL (or some appropriate extension of this logic) is a suitable formalism to specify properties of ASMs, then the verification problem for ASMs can be recasted as follows:

> Given an ASM M and a FBTL sentence φ over the vocabulary of M, decide whether for every input $\mathcal{I}$ appropriate for M, $C_M(\mathcal{I}) \models \varphi$.

Notice, however, that this problem is still **not** a computational problem. The difficulty here is that the initialization mapping of an ASM M may not be finitely representable, in which case M cannot serve as input to a computational device. Therefore, we focus our attention on classes of ASMs uniformly initialized in the sense of the next definition.

Definition 1.1.5. A class C of ASMs is *uniformly initialized* if for every ASM vocabulary Υ, all ASMs in C of vocabulary Υ have the same initialization mapping. □

With every uniformly initialized class C of ASMs and every ASM vocabulary Υ we can associate an initialization mapping $\mathit{initial}^C_\Upsilon$ such that $\mathit{initial}^C_\Upsilon$ is the initialization mapping of every $M \in C$ of vocabulary Υ. Observe that, in the context of a uniformly initialized class C of ASMs, every $M \in C$ is uniquely determined by its vocabulary and program. More precisely, if Υ is the vocabulary and Π the program of M, then necessarily $M = (\Upsilon, \mathit{initial}^C_\Upsilon, \Pi)$. This motivates the following definition.

Definition 1.1.6. The *standard representation* of an ASM $(\Upsilon, \mathit{initial}, \Pi)$ is given by the pair (Υ, Π). □

We can now define the verification problem for ASMs. Let C be a uniformly initialized class of ASMs and F a fragment of FBTL. The *verification problem for C and F* is the following decision problem:

VERIFY(C, F) : Given the standard representation of an ASM $M \in C$ and a sentence $\varphi \in F$ over the vocabulary of M, decide whether for every input $\mathcal{I}$ appropriate for M, $C_M(\mathcal{I}) \models \varphi$.

Remark 1.1.7. Note the subtlety in the above definition of VERIFY, namely that by viewing an ASM $M \in C$ as a finitely representable pair (Υ, Π) the problem of actually representing the initialization mapping of M has not been solved, but rather has been made part of the verification problem itself. □

For a systematic investigation of the automatic verifiability of ASMs, i.e., the decidability of the verification problem, it is reasonable to start with ASMs whose initialization mapping is particularly simple. To underline this, we provide below an example of a uniformly initialized class C of ASMs such that VERIFY$(C, \{accept\})$ is undecidable, although each single $M \in C$ is a very simple ASM.

Example 1.1.8. For the sake of simplicity, let us pretend that ASM vocabularies contain only input symbols and the distinguished boolean symbol *accept*, which is dynamic. Choose some undecidable problem

$$P \subseteq \{\Upsilon : \Upsilon \text{ is an ASM vocabulary}\}.$$

(For instance, let TM_Υ be the Turing machine whose encoding—in some fixed standard binary encoding—equals the binary representation of the number of symbols in Υ. Then $P := \{\Upsilon : TM_\Upsilon \text{ halts on the empty word}\}$ is undecidable, as an easy reduction of the halting problem for Turing machines shows.) For every ASM vocabulary Υ, set $M_\Upsilon = (\Upsilon, \mathit{initial}_\Upsilon, \mathit{accept} := \mathit{accept})$, where $\mathit{initial}_\Upsilon$ is such that for every input $\mathcal{I}$ appropriate for M_Υ

1. $\mathit{initial}(\mathcal{I})|_{\text{in}} = \mathcal{I}$ and
2. the interpretation of *accept* in $\mathit{initial}_\Upsilon(\mathcal{I})$ is *true* iff $\Upsilon \in P$.

Let C denote the class of all M_Υ. Now verify that $\Upsilon \mapsto (M_\Upsilon, \mathit{accept})$ is a reduction of P to VERIFY$(C, \{accept\})$. This shows that VERIFY$(C, \{accept\})$ is undecidable. □

A class of ASMs with particularly simple initialization mappings is the class of ASMs *finitely initialized* in the sense of the following definition. Finitely initialized ASMs will play an important role in the remainder of this chapter.

Definition 1.1.9. An ASM $M = (\Upsilon, \mathit{initial}, \Pi)$ is *finitely initialized* if for every input $\mathcal{I}$ appropriate for M, $\mathit{initial}(\mathcal{I})|_{\text{in}} = \mathcal{I}$ and in $\mathit{initial}(\mathcal{I})|_{\text{stat,dyn,out}}$, all relations are empty and all functions have range $\{0\}$. □

To see an example of a finitely initialized ASM, recall M_{reach} in Example 1.1.4. Note that every class of finitely initialized ASMs is in particular uniformly initialized.

1.2 Sequential Nullary ASMs

In this section, we introduce a class of restricted ASMs for which the verification problem will turn out to be decidable. Basically, the class consists of all finitely initialized ASMs in whose program

1. all dynamic symbols are nullary,
2. all guards are quantifier-free, and
3. `do-for-all` does not occur.

Observe that an ASM satisfying the last two conditions is sequential. In particular, the amount of work that can be accomplished by such an ASM in one computation step does not depend on the size of the input. In contrast, ASMs equipped with first-order quantification or parameterized parallel composition are capable of parallel computations. For instance, a guard of the form $\forall x \varphi(x)$ expresses a massively parallel search of the input universe. Due to the above syntactic restrictions, we call our ASMs *sequential nullary ASMs*.

Remark 1.2.1. Notice the analogy between nullary dynamic symbols and program variables: updating a nullary dynamic symbol v in one computation step corresponds to assigning a new value to the 'variable' v. Thus, sequential nullary ASMs can be viewed as non-deterministic algorithms equipped with no other means of dynamic storage than a fixed number of variables whose range is restricted to the input domain. □

Proviso. In the remainder of this chapter all ASMs are finitely initialized. □

To ease notation we will from now on do not distinguish between a finitely initialized ASM $(\Upsilon, \mathit{initial}, \Pi)$ and its standard representation (Υ, Π).

Definition 1.2.2. Fix an ASM vocabulary Υ where Υ_{dyn} contains only nullary symbols and $\Upsilon_{\text{stat}} = \Upsilon_{\text{out}} = \varnothing$. *Sequential nullary programs* are defined inductively:

- **Updates:** For every nullary relation symbol $b \in \Upsilon_{\text{dyn}}$, every nullary function symbol $v \in \Upsilon_{\text{dyn}}$, and every term t over Υ, each of the following is a sequential nullary program: b, $\neg b$, $v := t$.
- **Conditionals:** If Π is a sequential nullary program and φ is a quantifier-free formula over Υ, then (`if` φ `then` Π) is a sequential nullary program.
- **Parallel composition:** If Π_0 and Π_1 are sequential nullary programs, then $(\Pi_0 || \Pi_1)$ is a sequential nullary program.

- **Non-deterministic choice:** If Π is a sequential nullary program, $\bar{x}$ is a tuple of pairwise distinct variables, and φ is a quantifier-free formula over Υ such that the formula $\exists \bar{x} \varphi$ is finitely valid (see also Remark 1.2.3 below), then (`choose` $\bar{x} : \varphi$, Π) is a sequential nullary program.

The *free* and *bound variables* of a sequential nullary program are defined in the obvious way.

A *sequential nullary ASM* is a finitely initialized ASM whose program is a sequential nullary program (without free variables). □

Remark 1.2.3. (a) Finite validity of $\exists \bar{x} \varphi$ in the last program-formation rule of Definition 1.2.2 guarantees that one can always choose interpretations of the variables $\bar{x}$ so that the guard φ is satisfied. Note that φ may contain free variables others than $\bar{x}$, in which case finite validity of $\exists \bar{x} \varphi$ means finite validity of the sentence obtained from $\exists \bar{x} \varphi$ by replacing all free variables with new constant symbols.

(b) If for the reader's favorite guard φ the formula $\exists \bar{x} \varphi$ is not finitely valid, then one may try to replace (`choose` $\bar{x} : \varphi$, Π) with (`choose` $\bar{x} : \psi$, Π), where $\psi := \varphi \vee \bar{x} = \bar{0}$, and to detect 'invalid' choices of $\bar{0}$ inside Π. Given that all free variables of φ occur among $\bar{x}$, $\exists \bar{x} \varphi'$ is obviously finitely valid. □

As an example of a sequential nullary ASM, consider M_{reach} in Example 1.1.4. Recall that two ASM programs are called equivalent if they define the same transition relation. The next lemma provides a normal form for sequential nullary programs. The proof of the lemma is easy and omitted here.

Lemma 1.2.4. *For every sequential nullary program one can obtain in polynomial time an equivalent sequential nullary program of the form* (`choose` $\bar{x} : \varphi$, Π) *where* Π *does not contain* `choose` *(i.e.,* Π *is a deterministic program).*

Computational Power of Sequential Nullary ASMs

We compare the computational power of sequential nullary ASMs with the computational power of non-deterministic Turing machines. Let Υ be a vocabulary, let Q be a boolean query on $\mathrm{Fin}(\Upsilon)$, and let M be a sequential nullary ASM with input vocabulary Υ and a dynamic vocabulary that contains the boolean symbol *accept*. We say that M *accepts* a finite structure $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ iff $C_M(\mathcal{A}) \models \mathbf{EF}\mathit{accept}$ (i.e., in the computation graph of M on $\mathcal{A}$ there is a path from the initial state to a state where *accept* holds). We say that M *computes* Q if for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, M accepts $\mathcal{A}$ iff $\mathcal{A} \in Q$.

Lemma 1.2.5. *(1) A boolean query is definable in existential transitive-closure logic* (E+TC) *iff it is computable by a sequential nullary ASM. (2) On finite successor structures, sequential nullary ASMs compute precisely the class of* NLOGSPACE*-computable boolean queries.*

The crux in the proof of the lemma is the observation that the one-step semantics of any sequential nullary ASM can be expressed in terms of an existential first-order formula. Formally, we can show the following proposition.

Proposition 1.2.6. *Let $M = (\Upsilon, \Pi)$ be a sequential nullary ASM and let $v_1, \ldots, v_k$ be an enumeration of Υ_{dyn}. There exists an existential first-order formula $\chi_M(\bar{x}, \bar{x}')$ over Υ_{in} such that for every $\mathcal{A} \in \text{Fin}(\Upsilon_{\text{in}})$ and all interpretations $\bar{a}, \bar{a}'$ of $\bar{x}, \bar{x}'$ chosen from the universe of $\mathcal{A}$*

$$\mathcal{A} \models \chi_M[\bar{a}, \bar{a}'] \;\Leftrightarrow\; ((\mathcal{A}, \bar{a}), (\mathcal{A}, \bar{a}')) \in \mathit{Trans}_\Pi, \tag{1.1}$$

where $(\mathcal{A}, \bar{a})$ (resp. $(\mathcal{A}, \bar{a}')$) denotes the state over Υ whose input component is $\mathcal{A}$ and whose dynamic component is such that each v_i is interpreted as the i-th element in $\bar{a}$ (resp. $\bar{a}'$). χ_M can be obtained from M in polynomial time.

Proof. By Lemma 1.2.4, we can assume that $\Pi = (\texttt{choose}\ \bar{y} : \varphi,\ \Pi')$ for some deterministic program Π' in which $\bar{y}$ may occur free. Let Υ' be the ASM vocabulary $\Upsilon \cup \{\bar{y}\}$, where the variables $\bar{y}$ are now considered to be input constant symbols. We first prove the proposition for the deterministic sequential nullary ASM $M' := (\Upsilon', \Pi')$. More precisely, we define a quantifier-free formula $\chi_{M'}(\bar{x}, \bar{x}')$ over Υ'_{in} such that equivalence (1.1) holds for M', every $\mathcal{A} \in \text{Fin}(\Upsilon'_{\text{in}})$, and all k-tuples $\bar{a}$ and $\bar{a}'$ of elements of $\mathcal{A}$.

It is not hard to see that Π' is equivalent to an ASM program of the form $||_j \Pi'_j$, where each Π'_j is a rule of the form ($\texttt{if}\ \psi\ \texttt{then}\ r$) with a quantifier-free formula ψ over $\Upsilon'_{\text{in}} \cup \{\bar{v}\}$ as guard, and an atomic program of the form $v_i := t$ as right-hand side. (For simplicity, we view relational updates of the form b and $\neg b$ as functional updates $b := 1$ and $b := 0$, respectively.) $||_j \Pi'_j$ can be obtained from Π' in polynomial time. In the following we assume that $\Pi' = ||_j \Pi'_j$.

For each dynamic symbol $v_i \in \Upsilon'_{\text{dyn}}$, define a quantifier-free formula $\chi_i(\bar{x}, x'_i)$ over Υ'_{in} as follows. Let $(\psi_1, t_1), \ldots, (\psi_n, t_n)$ be an enumeration of all pairs (ψ_j, t_j) for which ($\texttt{if}\ \psi_j\ \texttt{then}\ v_i := t_j$) is a rule in Π'. For each such pair (ψ_j, t_j), set $\psi'_j = \psi_j[\bar{v}/\bar{x}]$ and $t'_j = t_j[\bar{v}/\bar{x}]$, and define:

$$\begin{aligned}
\mathit{conflict}_i(\bar{x}) &:= \bigvee\nolimits_{j,k} (\psi'_j \wedge \psi'_k \wedge t'_j \neq t'_k) \\
\chi_i(\bar{x}, x'_i) &:= \Big[\neg \mathit{conflict}_i \wedge \bigvee\nolimits_j (\psi'_j \wedge t'_j = x'_i)\Big] \;\vee \\
&\qquad \Big[\Big(\mathit{conflict}_i \vee \bigwedge\nolimits_j \neg\psi'_j\Big) \wedge x_i = x'_i\Big].
\end{aligned}$$

If we set $\chi_{M'}(\bar{x}, \bar{x}') = \bigwedge_i \chi_i(\bar{x}, x'_i)$, then equivalence (1.1) holds for M', every $\mathcal{A} \in \text{Fin}(\Upsilon'_{\text{in}})$, and all k-tuples $\bar{a}$ and $\bar{a}'$ of elements in $\mathcal{A}$.

The desired existential formula expressing the one-step semantics of M can now be defined as follows:

$$\chi_M(\bar{x}, \bar{x}') := \exists \bar{y} \big(\varphi' \wedge \chi_{M'}(\bar{y}, \bar{x}, \bar{x}')\big),$$

where φ' is obtained from φ be replacing every occurrence of v_i with x_i. χ_M is polynomial-time computable from M. □

Proof of Lemma 1.2.5. To (1). Consider a boolean query Q on $\mathrm{Fin}(\Upsilon)$. For the "if" direction suppose that Q is computable by a sequential nullary ASM M. Let $\chi_M(\bar{x}, \bar{x}')$ be obtained from M according to Proposition 1.2.6, where, w.l.o.g., we can assume that in the enumeration of the dynamic symbols of M, *accept* occurs first. As in the proof of Proposition 1.2.6, we view boolean dynamic relations as nullary dynamic functions ranging in $\{0, 1\}$. The following (E+TC) sentence over Υ defines Q:

$$\exists \bar{x}' \big([\mathrm{TC}_{\bar{x},\bar{x}'}\ \chi_M(\bar{x}, \bar{x}')](\bar{0}, \bar{x}') \wedge x'_1 = 1 \big).$$

For the "only if" direction suppose that Q is definable by an (E+TC) sentence φ over Υ. By the Normal Form Lemma in Chapter 3 (Lemma 3.1.4, see also the proof of Corollary 3.1.5) there exists a quantifier-free formula $\psi(\bar{x}, \bar{x}')$ over Υ such that φ is equivalent to $[\mathrm{TC}_{\bar{x},\bar{x}'}\ \psi](\bar{0}, \bar{1})$. We modify $M_{\mathrm{reach}} = (\Upsilon, \Pi_{\mathrm{reach}})$ in Example 1.1.4 to a sequential nullary ASM $M'_{\mathrm{reach}} = (\Upsilon', \Pi'_{\mathrm{reach}})$ computing Q. For each variable x_i in $\bar{x}$, introduce a new nullary function symbol p_i. (p_i is short for $pebble_i$.) Define Υ' by setting $\Upsilon'_{\mathrm{in}} = \Upsilon$, $\Upsilon'_{\mathrm{dyn}} = \{accept, running, \bar{p}\}$, and $\Upsilon'_{\mathrm{stat}} = \Upsilon_{\mathrm{out}} = \varnothing$. Obtain Π'_{reach} from Π_{reach} be replacing every occurrence of *pebble*, z, *source*, *target*, and $E(pebble, z)$ with $\bar{p}$, $\bar{z}$, $\bar{0}$, $\bar{1}$, and $\psi[\bar{x}/\bar{p}, \bar{x}'/\bar{z}]$, respectively. Then, $M'_{\mathrm{reach}} = (\Upsilon', \Pi'_{\mathrm{reach}})$ is a sequential nullary ASM computing Q.

The second assertion is an immediate consequence of (1) and well-known results from descriptive complexity theory [Imm87, EF95, Imm98]. □

ASMs with Only One Nullary Dynamic Function

We show that sequential nullary ASMs whose dynamic vocabulary consists of one (nullary) function symbol and arbitrarily many boolean symbols recognize precisely the regular languages.

For the sake of simplicity, let us focus on regular languages over the alphabet $\{0, 1\}$. Observe that any such language can be viewed as a boolean query on finite successor structures over the vocabulary $\{P, succ, 0\}$ where P is a set symbol. Let $\Upsilon := \{P, succ, 0\}$. There is a one-to-one correspondence between non-empty words over $\{0, 1\}$ and finite successor structures over Υ. A word $w_1 \dots w_n \in \{0, 1\}^+$ corresponds to $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ iff the cardinality of (the universe of) $\mathcal{A}$ is n, and for every $i \in \{1, \dots, n\}$

$$w_i = 1 \quad \Leftrightarrow \quad \mathcal{A} \models P(succ^i(0)).$$

This correspondence can be lifted to a one-to-one correspondence between languages over $\{0, 1\}$ (which do not contain the empty word) and boolean queries

on $\mathrm{Fin}(\Upsilon)$ in the obvious way. We call a boolean query Q on $\mathrm{Fin}(\Upsilon)$ *regular* if the language over $\{0,1\}$ corresponding to Q is regular.

Lemma 1.2.7. *Let Υ be as above. A boolean query on $\mathrm{Fin}(\Upsilon)$ is regular iff it is computable by a sequential nullary ASM whose dynamic vocabulary consists of one (nullary) function symbol and arbitrarily many boolean symbols.*

Proof. Consider a boolean query Q on $\mathrm{Fin}(\Upsilon)$. For the "only if" direction suppose that Q is regular. Let $L_Q \subseteq \{0,1\}^+$ be the language corresponding to Q. Since L_Q is regular, there exists a deterministic finite automaton $A = (Q', \{0,1\}, \delta, q_0, q_{acc})$ which accepts L_Q. We define a sequential nullary ASM $M_A = (\Upsilon_A, \Pi_A)$ that computes Q and uses exactly one dynamic function.

W.l.o.g., we can assume that the state set Q' of A is $\{q_{\bar{b}} : \bar{b} \in \{0,1\}^m\}$ and that $q_{\bar{0}} = q_0$ and $q_{\bar{1}} = q_{acc}$. To the definition of Υ_A. Since M_A is supposed to be a sequential nullary ASM, it suffices to specify the input and dynamic vocabulary of M_A. Let the input vocabulary be Υ and let the dynamic vocabulary be $\{pebble, accept, b_1, \ldots, b_m\}$ where *pebble* is a nullary function symbol and *accept* and $b_1, \ldots, b_m$ are boolean symbols. In the following, we write 0 and 1 instead of *false* and *true*, respectively. In particular, $(b_i := 0)$ and $(b_i := 1)$ stand for the relational updates b_i and $\neg b_i$, respectively. To the definition of Π_A. For every transition $t \in \delta$, if $t = (q_{\bar{e}}, in) \to q_{\bar{e}'}$, then set

$$\Pi_t \;=\; \texttt{if}\ (\bar{b} = \bar{e}) \wedge \varphi_{in}\ \texttt{then}\ \bar{b} := \bar{e}',$$

where $\varphi_{in} := P(pebble)$ if $in = 1$; otherwise, $\varphi_{in} := \neg P(pebble)$. Let Π_A be defined as follows:

```
program Π_A:
  ||_{t∈δ} Π_t
  pebble := succ(pebble)
  if (b̄ = 1̄) then accept
```

For the "if" direction suppose that Q is computable by a sequential nullary ASM M whose dynamic vocabulary contains only one function symbol and, say, m boolean symbols. Following the "if" direction in the proof of Lemma 1.2.5, we obtain the subsequent (E+TC) sentence over Υ defining Q:

$$\exists \bar{b}' \exists x' \big([\mathrm{TC}_{\bar{b}x, \bar{b}'x'}\ \chi_M(\bar{b}x, \bar{b}'x')](\bar{0}0, \bar{b}'x') \wedge b'_1 = 1 \big),$$

where χ_M is an existential first-order formula, $\bar{b}$ and $\bar{b}'$ are two m-tuples of boolean variables (each representing the m boolean dynamic symbols of M), and x and x' are two individual variables (each representing the dynamic function symbol of M). We show that there is a sentence definable in monadic second-order logic (MSO) [EF95] that is equivalent to the above (E+TC) sentence. Since on finite successor structures over monadic vocabularies a query is definable in MSO iff it is regular [Büc60], this will imply regularity of Q.

First observe that it suffices to provide for every TC formula φ of the form $[\mathrm{TC}_{\bar{b}x,\bar{b}'x'}\,\chi](\bar{0}t,\bar{1}t')$, where χ is a FO formula, an equivalent MSO formula. (Note that any TC formula can be replaced with an equivalent TC formula whose argument tuple has the form $(\bar{0},\bar{1})$; see, e.g., equivalence (1) in the proof of Lemma 3.1.4.) To ease notation, let us assume from here on that $m = 1$. In particular, we write b and b' instead of $\bar{b}$ and $\bar{b}'$, respectively. Verify that the following equivalences hold:

$$\begin{aligned}\varphi &\equiv \big[\mathrm{LFP}_{X,bx}\,(bx = 1t') \vee (\exists b'x' \in X)\chi(bx, b'x')\big](0t)\\ &\equiv \big[\text{S-LFP}_{X_0,x_0,\,X_1,x_1}\;\chi_0(X_0,X_1,x_0),\chi_1(X_0,X_1,x_1)\big](t)\end{aligned}$$

where the latter formula, called it φ', is a simultaneous least fixed-point formula with

$$\begin{aligned}\chi_0(X_0,X_1,x_0) &:= \bigvee\nolimits_{e\in\{0,1\}}(\exists x' \in X_e)\chi[b/0, x/x_0, b'/e]\\ \chi_1(X_0,X_1,x_1) &:= \bigvee\nolimits_{e\in\{0,1\}}(\exists x' \in X_e)\chi[b/1, x/x_1, b'/e] \vee (x_1 = t').\end{aligned}$$

Since both relation variables X_0 and X_1 are unary, φ' is equivalent to the following MSO formula:

$$\forall X_0X_1\Big[\Big(\bigwedge\nolimits_{e\in\{0,1\}}\forall x_e\big(x_e \in X_e \leftrightarrow \chi_e(X_0,X_1,x_e)\big)\Big) \to t \in X_0\Big].$$

□

Lemma 1.2.7 does not hold for sequential nullary ASMs equipped with two nullary dynamic functions. To see this, notice that by Lemma 1.2.7 the emptiness problem, i.e., the problem of deciding whether a given ASM accepts any input, is decidable for sequential nullary ASMs with only one nullary dynamic function. However, a result in the next section implies that this problem is undecidable for sequential nullary ASMs with two nullary dynamic functions (see Corollary 1.3.8). This shows that sequential nullary ASMs equipped with two nullary dynamic functions are strictly more powerful than those equipped with only one such function.

1.3 Verifiability Results

We can now present our results concerning the automatic verifiability of ASMs. We first show that the verification problem is decidable for sequential nullary ASMs with relational input and properties expressible in a rich fragment of FBTL. We then discuss three straightforward approaches to generalize this positive result and show that in each case the verification problem becomes undecidable. More precisely, we prove that, if unary functions are permitted (either in inputs or as

dynamic storage) or if first-order quantifiers occur in guards, then the verification problem is undecidable. Altogether our results in this section might suggest that with sequential nullary ASMs with relational input we are approaching the limit of automatic verifiability of ASMs. (See also the discussion at the end of Part I.)

Sequential Nullary ASMs with Relational Input

In favor of a succinct formulation of our main positive result we introduce some additional notation.

Definition 1.3.1. SN-ASM$_{\text{rel}}$ is the class of sequential nullary ASMs whose input vocabulary is relational. (Recall our convention that relational vocabularies may contain constant symbols.) □

The next definition introduces two fragments of the specification logic FBTL. We will show that sequential nullary ASMs with relational input can be verified against properties expressible in either fragment.

Definition 1.3.2. ETE denotes the set of FBTL formulas derivable by means of the following formula-formation rules:

(S1') Every atomic and negated atomic FO formula is a state formula.

(S2) If φ is a path formula, then $\mathbf{E}\varphi$ is a state formula.

(P1) Every state formula is a path formula.

(P2) If φ and ψ are path formulas, then $\mathbf{X}\varphi$, $\varphi\mathbf{U}\psi$, and $\varphi\mathbf{B}\psi$ are path formulas.

(SP1') If φ and ψ are state (resp. path) formulas, then $\varphi \vee \psi$ and $\varphi \wedge \psi$ are state (resp. path) formulas.

(SP2') If x is a variable and φ a state formula, then $\exists x\varphi$ is a state formula.

UTU denotes the set of negated ETE formulas. □

Finally, we define a restriction of the verification problem for ASMs. For every natural number m, let VERIFY$_m$ denote the restriction of VERIFY to instances where only symbols of arity at most m occur. An investigation of this restriction is motivated by the observation that the arities of relations and functions used in practice tend to be rather small. Indeed, for practical purposes it often suffices to solve VERIFY for ASMs whose vocabulary contains only symbols of arity $\leq m$, for some a priori fixed natural number m. The following theorem states our main positive result.

Theorem 1.3.3. *The following problem is* PSPACE*-complete for any $m \geq 0$:*

$$\text{VERIFY}_m(\text{SN-ASM}_{\text{rel}}, \text{ETE} \cup \text{UTU}). \tag{1.2}$$

In other words, verifying sequential nullary ASMs with relational input against (ETE ∪ UTU)*-specifications is a* PSPACE*-complete problem, given that the maximal arity of the employed input relations is a priori bounded.*

To prove the containment assertion of the theorem, i.e., that problem (1.2) is in PSPACE, we use the following observation. Recall that by FIN-VAL(L) (resp. FIN-SAT(L)) we denote the finite validity (resp. finite satisfiability) problem for a logic L.

Proposition 1.3.4. *Let C be a class of ASMs and let F be a fragment of* FBTL*. Suppose that there exists a logic L satisfying the following condition:*

> *There is a computable function that maps every instance (M, φ) of* VERIFY(C, F) *to an L-sentence $\chi_{M,\varphi}$ over the input vocabulary of M such that for every input $\mathcal{I}$ appropriate for M*
>
> $$C_M(\mathcal{I}) \models \varphi \quad \Leftrightarrow \quad \mathcal{I} \models \chi_{M,\varphi}. \tag{1.3}$$

(1) If FIN-VAL(L) *is decidable, then so is* VERIFY(C, F)*. (2) If* FIN-SAT(L) *is decidable, then so is* VERIFY$(C, \neg F)$*, where $\neg F$ denotes the set of negated F-sentences.*

Proof. By assumption, $(M, \varphi) \mapsto \chi_{M,\varphi}$ is a reduction from VERIFY(C, F) to FIN-VAL(L). This shows the first assertion. For the second assertion verify the following chain of equivalences: $(M, \neg\varphi) \in$ VERIFY$(C, \neg F)$ iff $C_M(\mathcal{I}) \models \neg\varphi$ for every $\mathcal{I}$ (appropriate for M) iff $C_M(\mathcal{I}) \not\models \varphi$ for every $\mathcal{I}$ iff $\mathcal{I} \not\models \chi_{M,\varphi}$ for every $\mathcal{I}$ iff $\chi_{M,\varphi} \notin$ FIN-SAT(L). Hence, $(M, \neg\varphi) \mapsto \chi_{M,\varphi}$ is a reduction from VERIFY$(C, \neg F)$ to the complement of FIN-SAT(L). Decidability of FIN-SAT(L) then implies the second assertion. □

Proof of Theorem 1.3.3. Containment: Set $C =$ SN-ASM$_{\text{rel}}$, $F =$ ETE, and $L =$ (E+TC), and recall that FIN-VAL$_m$ and FIN-SAT$_m$ denote the restrictions of FIN-VAL and FIN-SAT, respectively, to instances where only symbols of arity $\leq m$ occur. We provide a polynomial-time reduction from VERIFY$_m(C, F)$ to FIN-VAL$_m(L)$. By the proof of Proposition 1.3.4, this reduction also reduces VERIFY$_m(C, \neg F)$ to the complement of FIN-SAT$_m(L)$. Since both FIN-VAL$_m(L)$ and FIN-SAT$_m(L)$ are in PSPACE (see Corollaries 3.2.3 and 3.2.8), and PSPACE is closed under complementation, this will imply containment of problem (1.2) in PSPACE.

Consider an instance (M, φ) of VERIFY$_m(C, F)$ and assume that Υ is the vocabulary of M. We define an (E+TC) sentence $\chi_{M,\varphi}$ over Υ_{in} satisfying equivalence (1.3) in Proposition 1.3.4. (The definition closely follows a construction

due to Immerman and Vardi that was first presented in [IV97] as a translation of the branching temporal logic CTL* into (FO+TC).) Fix an enumeration of Υ_{dyn}, say, $v_1, \ldots, v_k$, and choose for each v_i a new variable x_i. For every state formula $\psi \in \mathrm{ETE}(\Upsilon)$ with $\mathrm{free}(\psi) = \{y_1, \ldots, y_m\}$ we are going to define a formula $\chi'_{M,\psi} \in (\mathrm{E+TC})(\Upsilon_{\mathrm{in}})$ with $\mathrm{free}(\chi'_{M,\psi}) = \{\bar{x}, \bar{y}\}$ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon_{\mathrm{in}})$ and all $(k+m)$-tuples $(\bar{a}, \bar{b})$ of elements of $\mathcal{A}$

$$\mathcal{A} \models \chi'_{M,\psi}[\bar{a}, \bar{b}] \quad \Leftrightarrow \quad (C_M(\mathcal{A}), (\mathcal{A}, \bar{a})) \models \psi[\bar{b}], \tag{1.4}$$

where $(\mathcal{A}, \bar{a})$ denotes the state in $C_M(\mathcal{A})$ whose dynamic component is such that each v_i is interpreted as the i-th element in $\bar{a}$. We can then define the desired (E+TC) sentence $\chi_{M,\varphi}$ to be $\chi'_{M,\varphi}[\bar{x}/\bar{0}]$.

By induction on the construction of ψ. The cases where ψ is obtained by means of the formula-formation rules **(S1')**, **(SP1')**, and **(SP2')** are straightforward. Suppose that ψ is obtained by means of rule **(S2)**, i.e., $\psi = \mathbf{E}\alpha$ for some path formula α. Let $\mathrm{cl}(\alpha)$ denote the set of those path subformulas of α whose occurrence is not strictly inside some state subformula of α. (Note that α can be built from the state formulas in $\mathrm{cl}(\alpha)$ by means of disjunction, conjunction, and the temporal operators $\mathbf{X}$, $\mathbf{U}$, and $\mathbf{B}$.) For every $\theta \in \mathrm{cl}(\alpha)$ introduce a boolean variable b_θ. If θ has the form $\theta_0 \mathbf{U} \theta_1$ then introduce an additional boolean variable m_θ different from b_θ. Let $\bar{b}$ (resp. $\bar{m}$) denote a tuple consisting of the boolean variables b_θ (resp. m_θ) in some random but fixed order.

Obtain the existential first-order formula $\chi_M(\bar{x}, \bar{x}')$ over Υ_{in} according to Proposition 1.2.6. For every state formula $\theta(\bar{y}) \in \mathrm{cl}(\alpha)$, let $\chi'_{M,\theta}(\bar{x}, \bar{y})$ be obtained from $\theta(\bar{y})$ by induction hypothesis. Define $\mathit{next} \in (\mathrm{E+TC})(\Upsilon_{\mathrm{in}})$ with $\mathrm{free}(\mathit{next}) \subseteq \{\bar{x}\bar{b}\bar{m},\ \bar{x}'\bar{b}'\bar{m}',\ \bar{y}\}$ as follows:

$$\mathit{next}(\bar{x}\bar{b}\bar{m}, \bar{x}'\bar{b}'\bar{m}', \bar{y}) \quad := \quad \chi_M(\bar{x}, \bar{x}') \wedge \bigwedge\nolimits_{\theta \in \mathrm{cl}(\alpha)} \mathit{next}_{M,\theta}(\bar{x}, \bar{y}, \bar{b}\bar{m}, \bar{b}'\bar{m}'),$$

where $\mathit{next}_{M,\theta}(\bar{x}, \bar{y}, \bar{b}\bar{m}, \bar{b}'\bar{m}')$ is

$$\begin{array}{ll}
b_\theta \to \chi'_{M,\theta}(\bar{x}, \bar{y}), & \text{if } \theta \text{ is a state formula} \\
b_\theta \to (b_{\theta_0} \vee (\wedge)\, b_{\theta_1}), & \text{if } \theta = \theta_0 \vee (\wedge)\, \theta_1 \\
b_\theta \to b'_{\theta_0}, & \text{if } \theta = \mathbf{X}\theta_0 \\
\big(b_\theta \to (b_{\theta_1} \vee (b_{\theta_0} \wedge b'_\theta))\big) \wedge \big(m'_\theta \to (m_\theta \vee b_{\theta_1})\big), & \text{if } \theta = \theta_0 \mathbf{U} \theta_1 \\
b_\theta \to (b_{\theta_1} \wedge (b_{\theta_0} \vee b'_\theta)), & \text{if } \theta = \theta_0 \mathbf{B} \theta_1.
\end{array}$$

Finally, let

$$\begin{aligned}
\chi'_{M,\psi}(\bar{x}, \bar{y}) \quad := \quad \exists \bar{b}, \bar{x}'\bar{b}', \bar{m}' \Big(& [\mathrm{TC}_{\bar{x}\bar{b}\bar{m}, \bar{x}'\bar{b}'\bar{m}'}\ \mathit{next}](\bar{x}\bar{b}\bar{0}, \bar{x}'\bar{b}'\bar{0}) \ \wedge \\
& [\mathrm{TC}^S_{\bar{x}\bar{b}\bar{m}, \bar{x}'\bar{b}'\bar{m}'}\ \mathit{next}](\bar{x}'\bar{b}'\bar{0}, \bar{x}'\bar{b}'\bar{m}') \ \wedge \\
& b_\alpha \wedge \bigwedge\nolimits_{\theta_0 \mathbf{U} \theta_1 \in \mathrm{cl}(\alpha)} (b'_{\theta_0 \mathbf{U} \theta_1} \to m'_{\theta_0 \mathbf{U} \theta_1}) \Big),
\end{aligned}$$

where TC^S denotes the strict version of the transitive closure operator TC. (TC^S is definable in (FO+TC), e.g., let

$$[\mathrm{TC}^S_{\bar{x},\bar{x}'}\ \psi](\bar{t},\bar{t}') \quad := \quad [\mathrm{TC}_{b,\bar{x},b'\bar{x}'}\ b'=1 \wedge \psi](0\bar{t},1\bar{t}'), \tag{1.5}$$

where b and b' are new boolean variables not occurring in ψ nor among $\bar{x},\bar{x}'$.)

It is not hard to verify that $\chi'_{M,\psi}$ indeed satisfies equivalence (1.4). We omit the details. It remains to show that $\chi'_{M,\psi}$ can be obtained from M and ψ in polynomial time. Again, the only interesting case is where ψ has the form $\mathbf{E}\alpha$. Suppose that for every state formula $\theta \in \mathrm{cl}(\alpha)$, $\chi'_{M,\theta}$ can be obtained from M and θ in polynomial time. It is easy to see that *next* is polynomial-time constructible. (Recall that by Proposition 1.2.6, $\chi_M(\bar{x},\bar{x}')$ can be obtained from M in polynomial time.) Unfortunately, $\chi'_{M,\psi}$ as defined above may not be polynomial-time constructible. This is because *next* occurs twice in the definition of $\chi'_{M,\psi}$, possibly causing the length of $\chi'_{M,\psi}$ to grow exponentially in the length of ψ. However, the exponential blow-up can be avoided by 'merging' the occurring TC subformulas. To see this, verify the following equivalence:

$$[\mathrm{TC}_{\bar{y},\bar{y}'}\ \mathit{next}](\bar{s},\bar{s}') \wedge [\mathrm{TC}^S_{\bar{y},\bar{y}'}\ \mathit{next}](\bar{t},\bar{t}') \quad \equiv \quad [\mathrm{TC}_{b_1b_2\bar{z},b'_1b'_2\bar{z}'}\ \chi](00\bar{s},11\bar{t}'),$$

where b_1, b'_1, b_2, b'_2 are new boolean variables, each z_i (resp. z'_i) is a new variable replacing y_i (resp. y'_i), and

$$\begin{aligned}\chi \quad := \quad & \Big(\big[(b_1=0 \wedge \bar{z} \neq \bar{s}' \wedge b'_1=0) \vee \\ & \qquad (b_1=1 \wedge \bar{z} \neq \bar{t}' \wedge b'_1=1)\big] \wedge (\mathit{next}[\bar{y}/\bar{z},\bar{y}'/\bar{z}'] \wedge b'_2=1)\Big) \ \vee \\ & (b_1=0 \wedge \bar{z}=\bar{s}' \wedge b'_1=1 \wedge b'_2=0 \wedge \bar{z}'=\bar{t}).\end{aligned}$$

(See also equivalence (7) in the proof of Lemma 3.1.4.) Hence, we can replace in the definition of $\chi'_{M,\psi}$ the occurrence of the left-hand side of the above equivalence with the corresponding right-hand side. The new definition of $\chi'_{M,\psi}$ is in polynomial time.

Hardness: Let C and L be as before and set $F = \{\mathbf{EF}\mathit{accept}\}$. We display two polynomial-time reductions, the first from $\textsc{fin-val}_m(L)$ to $\textsc{verify}_m(C,F)$ and the second from $\textsc{fin-sat}_m(L)$ to the complement of $\textsc{verify}_m(C,\neg F)$. This will show PSPACE-hardness of both $\textsc{verify}_m(C,F)$ and $\textsc{verify}_m(C,\neg F)$ because $\textsc{fin-val}_m(L)$ and $\textsc{fin-sat}_m(L)$ are PSPACE-hard (see Corollaries 3.2.3 and 3.2.8) and PSPACE is closed under complementation.

Consider an instance φ of $\textsc{fin-val}_m(L)$. Following the "only if" direction in the proof of Lemma 1.2.5, we obtain in polynomial time a sequential nullary ASM M'_{reach} computing the boolean query defined by φ. One easily verifies that $\varphi \mapsto (M'_{\mathrm{reach}}, \mathbf{EF}\mathit{accept})$ is a reduction from $\textsc{fin-val}_m(L)$ to $\textsc{verify}_m(C,F)$. The same line of thought shows that $\varphi \mapsto (M'_{\mathrm{reach}}, \mathbf{AG}\neg\mathit{accept})$ is a reduction from $\textsc{fin-sat}_m(L)$ to the complement of $\textsc{verify}_m(C,\neg F)$. □

Corollary 1.3.5. VERIFY(SN-ASM$_{\text{rel}}$, ETE $\cup$ UTU) $\in$ EXPSPACE.

Proof. The reduction in the first part of the proof of Theorem 1.3.3 also reduces VERIFY(SN-ASM$_{\text{rel}}$, ETE) to FIN-VAL(E+TC), and VERIFY(SN-ASM$_{\text{rel}}$, UTU) to the complement of FIN-SAT(E+TC). Since FIN-VAL(E+TC) and FIN-SAT(E+TC) are in EXPSPACE (see Corollaries 3.2.3 and 3.2.8), and EXPSPACE is closed under complementation, this implies the corollary. □

On Inputs with Functions

We now consider sequential nullary ASMs whose input vocabulary contains (non-nullary) function symbols. It turns out that most basic safety and liveness properties are undecidable for such ASMs.

Definition 1.3.6. SN-ASM$_{\text{fct}}$ is the class of sequential nullary ASMs whose input vocabulary contains two non-nullary symbols, one of which is a function symbol. □

A minimal requirement on any automatic verifier for ASMs is that, when given an ASM M, it should be able to decide whether M *can reach a safe state* on every input, or, equally desirable, whether M *reaches only safe states* on every input. Here, safety for a state could mean that some distinguished boolean variable assumes a particular truth value. This motivates the following definition of two simple verification problems. Let C be a class of (finitely initialized) ASMs.

LIVENESS(C) : Given the standard representation of an ASM $M \in C$ and a boolean dynamic symbol b in the vocabulary of M, decide whether for every input $\mathcal{I}$ appropriate for M, $C_M(\mathcal{I}) \models \mathbf{EF}b$.

Let SAFETY(C) be defined as LIVENESS(C), except that $\mathbf{EF}b$ is replaced with $\mathbf{AG}\neg b$.

Theorem 1.3.7. *The problems* LIVENESS(SN-ASM$_{\text{fct}}$) *and* SAFETY(SN-ASM$_{\text{fct}}$) *are both undecidable.*

Proof. Set C = SN-ASM$_{\text{fct}}$ and L = E+TC. As in the second part of the proof of Theorem 1.3.3, we reduce FIN-VAL(L) to LIVENESS(C), and FIN-SAT(L) to the complement of SAFETY(C). The theorem is then implied by Theorem 3.4.1. Consider an instance φ of FIN-VAL(L). Follow the "only if" direction in the proof of Lemma 1.2.5 to obtain a sequential nullary ASM M'_{reach} that computes the boolean query defined by φ. Then $\varphi \mapsto (M'_{\text{reach}}, \mathit{accept})$ is a reduction from FIN-VAL(L) to LIVENESS(C). The same mapping reduces FIN-SAT(L) to the complement of SAFETY(C). □

A refinement of the last proof yields the following corollary.

Corollary 1.3.8. *If C includes all deterministic ASMs in* SN-ASM$_{\text{fct}}$ *whose dynamic vocabulary consists of two (nullary) function symbols and arbitrarily many boolean symbols, then* SAFETY(C) *is undecidable.*

Proof. We follow the proof of Theorem 3.4.1 and reduce the undecidable emptiness problem for deterministic finite 2-head automata to SAFETY(C) (see Lemma 3.4.3). Consider a 2-head DFA $A = (Q, \Sigma, \delta, q_0, q_{acc})$ as in the proof of Theorem 3.4.1. We define a sequential nullary ASM $M_A \in C$ such that

$$(M_A, \mathit{accept}) \in \text{SAFETY}(C) \quad \Leftrightarrow \quad L(A) = \varnothing.$$

The construction of $M_A = (\Upsilon_A, \Pi_A)$ closely follows the construction of M_A in the "only if" direction of the proof of Lemma 1.2.7. In fact, Υ_A is as in the proof of the lemma, except that it now contains a unary input function symbol f (instead of *succ*) and two nullary dynamic function symbols pebble_1 and pebble_2 (instead of *pebble*). Π_A is defined as follows:

```
program Π_A:
  ||_{t∈δ} Π_t
  if (b̄ = 1̄) then accept
```

where for every transition $t = (q_{\bar{e}}, \mathit{in}_1, \mathit{in}_2) \to (q_{\bar{e}'}, \mathit{move}_1, \mathit{move}_2)$ in δ

$$\Pi_t \quad := \quad \texttt{if } (\bar{b} = \bar{e}) \wedge \alpha_1' \wedge \alpha_2' \texttt{ then } (\bar{b} := \bar{e}' || \beta_1' || \beta_2'),$$

and α_1', α_2', β_1', and β_2' denote the obvious modifications of α_1, α_2, β_1, and β_2 in the proof of Theorem 3.4.1. □

The corollary does not necessarily hold for classes of sequential nullary ASMs equipped with only one nullary dynamic function. As a counterexample recall Lemma 1.2.7. The following two observations are the crux in the proof of the corollary:

1. Input structures that contain non-nullary functions can serve as bit-string encodings.

2. ASMs equipped with (at least) two nullary dynamic functions are powerful enough to check whether a bit-string (encoded in an input structure) represents an accepting computation of a Turing machine.

Indeed, the corollary remains valid if we redefine SAFETY as follows. Let K denote a class of input structures (over some fixed input vocabulary Υ_{in}) such that for every bit-string s there exists an input structure $\mathcal{I}_s \in K$ encoding s, say, by means of a constant 0 (marking the beginning of s), a binary relation E (serving as successor relation), and a set P (representing bits). Redefine SAFETY so that the input of a given sequential nullary ASM (with input vocabulary Υ_{in}) ranges in K rather than in the class of all input structures appropriate for the

ASM. If C is a class of ASMs as in Corollary 1.3.8, then SAFETY(C) is still undecidable. Altogether this indicates that even very simple ASMs cannot be verified automatically if their input structures can serve as bit-string encodings.

More Powerful ASMs

In view of the negative result for ASMs whose input contain functions, we now return to ASMs with relational input. We investigate the decidability of SAFETY for two classes of generalized sequential nullary ASMs. The first class consists of sequential nullary ASMs equipped with one unary dynamic function. Note that a unary dynamic function can be viewed as a one-dimensional array. The second class consists of sequential nullary ASMs in whose program a single first-order quantifier may occur in one guard. A first-order quantifier introduces a restricted form of parallelism.

Definition 1.3.9. (SN-ASM+array) is the class of sequential nullary ASMs defined according to the following modification of Definition 1.2.2. Allow the ASM vocabulary Υ to contain one unary dynamic function symbol f. Furthermore, modify the first program-formation rule so that now updates of the form $f(t') := t$ can be derived.

(SN-ASM+quantifier) is the class of sequential nullary ASMs defined according to the following modification of Definition 1.2.2. Add a new program-formation rule that introduces conditionals whose guard may contain a single first-order quantifier, and require that this new rule is applied at most once while deriving a program. □

By (SN-ASM$_{\text{rel}}$+array) and (SN-ASM$_{\text{rel}}$+quantifier) we denote the restrictions of (SN-ASM+array) and (SN-ASM+quantifier), respectively, to ASMs with relational input vocabulary.

Lemma 1.3.10. *Let P be a non-nullary relation symbol. If C includes all ASMs in* (SN-ASM$_{\text{rel}}$+array) *with input vocabulary $\{P, 0, 1\}$, then* SAFETY(C) *is undecidable.*

Proof. We modify the proof of Corollary 1.3.8 and reduce the emptiness problem for 2-head DFAs to SAFETY(C). For every (appropriate) 2-head DFA A let the sequential nullary ASM M_A be defined as in the proof of Corollary 1.3.8. The idea is as follows. In a pre-computation, M_A randomly chooses a path from node 0 to node 1 in the current input structure and stores this path in the unary dynamic function f. Then, M_A proceeds as before, now using f as input function. Obtain $M'_A \in C$ from M_A by replacing the program Π_A of M_A with

`program` Π'_A:

  `if` $\neg$*running* `then`

```
    if pebble ≠ 1 then
       choose z : true
          f(pebble) := z
          pebble := z
    else
       f(pebble) := pebble
       running := true

 if running then
    Π_A
```

One can show that $L(A) \neq \varnothing$ iff there exists an input $\mathcal{I}$ appropriate for M'_A such that $C_{M'_A}(\mathcal{I}) \models \mathbf{EF}\, accept$. □

Lemma 1.3.11. *Let E be a relation symbol of arity ≥ 2. If C includes all ASMs in* (SN-ASM$_{\text{rel}}$+quantifier) *with input vocabulary $\{E, 0, 1\}$, then* SAFETY(C) *is undecidable.*

Proof. W.l.o.g., we can assume that E is binary. We reuse the proof of the previous lemma. The only difference is that, instead of randomly choosing a path, M'_A now searches the input structure for a path from node 0 to node 1 composed from E-edges. During the search, M'_A checks whether this path is deterministic in the sense that every node a on the path has at most two E-successors, namely the next node on the path and possibly a itself. Π'_A in the proof of the previous lemma is modified as follows:

```
program Π'_A:

 if ¬running then
    if pebble ≠ 1 then
       choose z : true
          if z ≠ pebble ∧ E(pebble, z) ∧
             ∀z'([z' ≠ pebble ∧ E(pebble, z')] → z' = z) then
             pebble := z
    else
       running := true

 if running then
    Π*_A
```

where Π^*_A is obtained from Π_A in the proof of Corollary 1.3.8 by replacing each rule Π_t, where $t = (q_{\bar{e}}, in_1, in_2) \to (q_{\bar{e}'}, move_1, move_2)$, with the following program:

```
program Π*_t:
 if (b̄ = ē) ∧ α*_1 ∧ α*_2 then
    choose y_1, y_2 : true
```

$$
\begin{array}{l}
\texttt{if } \beta_1^* \wedge \beta_2^* \texttt{ then} \\
\quad \bar{b} := \bar{e}' \\
\quad pebble_1 := y_1 \\
\quad pebble_2 := y_2
\end{array}
$$

where

$$
\alpha_i^* \quad := \quad \begin{cases} pebble_i \neq 1 \wedge E(pebble_i, pebble_i) & \text{if } in_i = 1 \\ pebble_i \neq 1 \wedge \neg E(pebble_i, pebble_i) & \text{if } in_i = 0 \\ pebble_i = 1 & \text{if } in_i = \varepsilon \end{cases}
$$

$$
\beta_i^* \quad := \quad \begin{cases} E(pebble_i, y_i) & \text{if } move_i = +1 \\ pebble_i = y_i & \text{if } move_i = 0. \end{cases}
$$

□

The last two lemmas show that, if the computational power of sequential nullary ASMs with relational input is increased in a straightforward manner, then most basic safety and liveness properties of the resulting ASMs can not be verified automatically.

2

Verification of Relational Transducers for Electronic Commerce

One of the main reasons for the enormous and still accelerating growth of the World Wide Web is the strong interest of commercial enterprises in electronic commerce, i.e., in offering services on and conducting business via the Web. Electronic commerce offers challenges to various disciplines of computer science, such as cryptography (for security and authentication), database systems (to support electronic transactions), and formal methods (for the design and verification of transaction protocols) [AY96, AVFY98]. In this chapter, we study the automatic verifiability of transaction protocols specifying the interaction of multiple parties via a network. The transaction protocols which we are concerned with typically occur in the context of electronic commerce applications, where they are also called *business models* [AVFY98]. As an example, consider an electronic warehouse where several customers interact with a supplier via the Internet. In this scenario the transaction protocol (or business model) of the supplier specifies how to react to requests of customers. Possible actions may include sending bills to customers, initiating delivery of products, updating the database of the supplier, etc.

As a general framework to formalize business models, Abiteboul, Vianu, Fordham, and Yesha have recently put forward *relational transducers* [AVFY98]. A relational transducer can be viewed as an interactive computational device equipped with an active database. In every computation step, such a transducer receives from its environment a collection of input relations (e.g., lists of orders and payments from various customers) and reacts by producing a collection of output

relations (e.g., lists of bills and items to be sent to customers). During a computation step, the transducer may also update its internal database, which is a relational database consisting of a dynamic and a static component. The dynamic component contains all temporary data necessary to keep track of ongoing transactions (e.g., a list of the currently ordered items). We refer to this component as the *memory* of the transducer. The purpose of the static component is to provide all static data, i.e., data which does not change during a transaction (e.g., the catalog of a supplier). We call this component the *database* of the transducer.

In [AVFY98], the authors have also investigated (and partially solved) several verification problems concerning relational transducers, among them:

(1) the problem of **verifying temporal properties** of relational transducers ("Does a business model meet its specification?"),

(2) the **log validation** problem ("Does a transaction, possibly carried out on a remote customer site, actually conform to the business model of a supplier?"), and

(3) the problem of **deciding equivalence** of relational transducers ("Does a business model that was customized by a business partner still conform to the original business model? That is, are two business models equivalent with respect to transactions?").

Although each of these problems is undecidable in general, positive results, i.e., decidability of variants of these problems, have been obtained for a class of restricted relational transducers, called *Spocus transducers* [AVFY98]. ("Spocus" is an acronym for "*S*emi-*p*ositive *o*utput and *cu*mulative *s*tates".) A drawback of Spocus transducers is that their memory relations (i.e., the dynamic relations of their internal storage) are *cumulative*: the memory of a Spocus transducer only accumulates all previous inputs; it cannot be updated otherwise. This limits the field of application of Spocus transducers substantially. Imagine, for example, a supplier who wants to enable customers to order a product multiple times during a session. The business model of this supplier cannot be expressed by means of a Spocus transducer, because once an order for a product has been placed, that order remains in the memory of a Spocus transducer until the end of the current session (see also Example 2.1.3).

Here, we investigate the decidability of the above verification problems for relational transducers more powerful than Spocus transducers. Our transducers, which we call *ASM (relational) transducers*, are defined by simple, rule-based ASM programs. ASM transducers are more powerful than Spocus transducers due to the following two reasons. Firstly, rule applications are in general guarded by first-order formulas. In contrast, Spocus transducers are defined by semi-positive datalog rules, i.e., rules guarded only by conjunctive formulas (built from atomic and negated atomic formulas). Secondly, the memory relations of

ASM transducers are not necessarily cumulative. Indeed, the memory of an ASM transducer can be seen as an active database with immediate triggering of insertion and deletion actions [PV98, AHV95]. As evidence for the computational power of ASM transducers, we show that they compute precisely the class of PSPACE-computable queries on ordered databases.

Since none of the verification problems (1)–(3) is, in its full generality, decidable for ASM transducers, we focus our attention on natural restrictions of these problems. The restrictions we impose are natural in the sense that they occur often in electronic commerce applications such that solutions of the restricted problems are still of practical importance. More precisely, we show that the verification problems are decidable for ASM transducers if one of the following two conditions is satisfied:

- **The (static) database relations of an ASM transducer are known.** This restriction is applicable if, say, the catalog of a supplier does not change frequently such that it becomes feasible to adjust verification to the currently valid catalog.

- **The maximal input flow which an ASM transducer is exposed to is a priori limited.** The *maximal input flow* is, roughly speaking, the maximal amount of input data forwarded to a relational transducer in one computation step. This restriction should be applicable in most electronic commerce applications. It is motivated by the observation that a relational transducer running in a realistic environment never receives 'too much' input in a single computation step, due to physical and technical limitations of the environment. For instance, the maximal input flow of a transducer running on an Internet server is limited by the number of clients accepted on the server, the capacity of the local network, the one-step capacity of the server, etc.

Under the first condition we obtain decidability for the class of all ASM transducers. Under the second condition decidability is obtained only for a class of restricted ASM transducers, called *ASM transducers with input-bounded quantification*. Transducers of the latter kind are weaker than general ASM transducers because first-order quantification in the guards of their rules is bounded to the active domain of the current input.

As it turns out, the maximal arity of the relations employed by ASM transducers is a major source for the complexity of the verification problems. We show that, if the maximal arity of the employed relations is a priori bounded, then all our restricted problems are PSPACE-complete. Imposing an upper bound on the maximal arity should be no serious obstacle for real-life applications, since the arities of relations used in practice tend to be rather small. On the other hand, if there is no fixed upper bound on the arities of relations, then almost all our restricted problems become EXPSPACE-complete.

Related Work. The general model of relational transducers as well as the verification problems we consider in this chapter are adopted from [AVFY98]. ASM relational transducers can be regarded as ASMs with external relations (see Remark 2.1.4 (b)). Applications of ASMs to database theory can be found in [GKS91] and [FAY97]. The latter work employs ASMs to specify active databases specially tailored for electronic commerce applications. Runs of ASM transducers are temporal databases in the spirit of [AHVdB96]. We use first-order temporal logic as defined in [Eme90] to express properties of runs of relational transducers (see problem (1) above). Our decidability results are implied by a reduction to the finite satisfiability problem for existential transitive closure logic. The reduction borrows elements of the Immerman-Vardi translation that was already employed in Chapter 1 (see the proof of Theorem 1.3.3). Our hardness results follow from results in [Var82, Var95, DEGV97].

Outline of the Chapter. In Sections 2.1 and 2.2, we provide a self-contained definition of ASM relational transducers and formally define the verification problems (1)–(3). Since none of the verification problems is decidable for the class of all ASM transducers, we propose natural restrictions of the problems in Section 2.3. In Section 2.4, we present our results concerning the verifiability of ASM transducers. Section 2.5 contains sketches of the proofs of these results.

2.1 ASM Relational Transducers

We consider relational transducers as defined in [AVFY98], though we will not presuppose any familiarity with that paper. We start with a self-contained definition of a powerful model of relational transducers based on ASMs. Our transducers, which we call *ASM (relational) transducers*, are defined by simple, rule-based ASM programs.

ASM transducer programs. A *transducer vocabulary* Υ is a quintuple (Υ_{in}, Υ_{db}, Υ_{mem}, Υ_{out}, Υ_{log}) of finite relational vocabularies where the first four vocabularies are pairwise disjoint, $\Upsilon_{\text{log}} \subseteq \Upsilon_{\text{in}} \cup \Upsilon_{\text{out}}$, and Υ_{in}, Υ_{mem}, and Υ_{out} do not contain constant symbols. For technical reasons we shall always assume that Υ_{db} contains (at least) the two constant symbols 0 and 1. The symbols in Υ_{in}, Υ_{db}, Υ_{mem}, Υ_{out}, and Υ_{log} are called *input*, *database*, *memory*, *output*, and *log symbols*, respectively. As in the case of ASM vocabularies, we sometimes overload notation and write Υ instead of $\Upsilon_{\text{in}} \cup \Upsilon_{\text{db}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$. The intended meaning will be clear from the context.

Definition 2.1.1. Let Υ be a transducer vocabulary. An *ASM transducer program* Π over Υ is a finite set of rules of the form

$$\texttt{if } \varphi(\bar{x}) \texttt{ then } (\neg)R(\bar{x})$$

where $\varphi(\bar{x})$ is a FO formula over $\Upsilon_{\text{in}} \cup \Upsilon_{\text{db}} \cup \Upsilon_{\text{mem}}$ with $\text{free}(\varphi) = \{\bar{x}\}$, and $R(\bar{x})$ is an atomic FO formula over $\Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$. If $R \in \Upsilon_{\text{out}}$, then $R(\bar{x})$ must occur positively on the right-hand side of the rule. $\varphi(\bar{x})$ is called the *guard* of the rule. □

The semantics of an ASM transducer program Π is similar to the one-step semantics of a Datalog($\neg\neg$) program [AHV95], except that Π treats inconsistent updates as 'no operations'.

Semantics. Let Υ be a transducer vocabulary and let Π be an ASM transducer program over Υ. Υ and Π define a relational transducer $T_{(\Upsilon,\Pi)}$ as follows.

A *state* $\mathcal{S}$ over Υ (of the relational transducer $T_{(\Upsilon,\Pi)}$) is a finite structure over the relational vocabulary Υ. As in the case of ASM states, we regularly write $\mathcal{S}|_{\text{db}}$ instead of $\mathcal{S}|\Upsilon_{\text{db}}$ (i.e., the reduct of $\mathcal{S}$ to vocabulary Υ_{db}) and $\mathcal{S}|_{\text{db,mem}}$ instead of $\mathcal{S}|(\Upsilon_{\text{db}} \cup \Upsilon_{\text{mem}})$, and so forth. A state $\mathcal{S}$ can be viewed as being composed of the four components $\mathcal{S}|_{\text{in}}$, $\mathcal{S}|_{\text{db}}$, $\mathcal{S}|_{\text{mem}}$, and $\mathcal{S}|_{\text{out}}$. Intuitively, $\mathcal{S}|_{\text{in}}$ is the current *input* of $T_{(\Upsilon,\Pi)}$ in state $\mathcal{S}$, while $\mathcal{S}|_{\text{out}}$ is the *output* that was produced during the last computation step of $T_{(\Upsilon,\Pi)}$. $\mathcal{S}|_{\text{db}}$ and $\mathcal{S}|_{\text{mem}}$ hold the *database* and the *memory* of $T_{(\Upsilon,\Pi)}$ in state $\mathcal{S}$, respectively.

The *relational transducer* $T_{(\Upsilon,\Pi)}$ (defined by Υ and Π) is a mapping from states over Υ to finite structures over $\Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$. If $\mathcal{S}$ is a state over Υ, then $T_{(\Upsilon,\Pi)}(\mathcal{S})$ determines the memory and the output of the transducer $T_{(\Upsilon,\Pi)}$ in a successor state $\mathcal{S}'$ of $\mathcal{S}$. The new input of $T_{(\Upsilon,\Pi)}$ in state $\mathcal{S}'$ is provided by the environment. The database of $T_{(\Upsilon,\Pi)}$ in $\mathcal{S}'$ is the same as in $\mathcal{S}$. We come to the definition of $T_{(\Upsilon,\Pi)}$. For simplicity, suppose that

- whenever $(\neg)R(\bar{x})$ is the right-hand side of a rule in Π, then the variable tuple $\bar{x}$ consists of pairwise distinct variables, and
- whenever $(\neg)R(\bar{x})$ and $(\neg)R'(\bar{y})$ are the right-hand sides of two rules in Π, and $R = R'$, then the variable tuples $\bar{x}$ and $\bar{y}$ are identical.

For each $R \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$, define the following FO formulas:

$$\varphi_R(\bar{x}) := \bigvee \big\{\varphi(\bar{x}) : \big(\texttt{if } \varphi(\bar{x}) \texttt{ then } R(\bar{x})\big) \in \Pi\big\} \tag{2.1}$$

$$\psi_R(\bar{x}) := \bigvee \big\{\varphi(\bar{x}) : \big(\texttt{if } \varphi(\bar{x}) \texttt{ then } \neg R(\bar{x})\big) \in \Pi\big\} \tag{2.2}$$

$$\begin{aligned}\chi_R(\bar{x}) := {} & \big(\varphi_R(\bar{x}) \wedge \neg\psi_R(\bar{x})\big) \vee \\ & \big(\varphi_R(\bar{x}) \wedge \psi_R(\bar{x}) \wedge R(\bar{x})\big) \vee \\ & \big(\neg\varphi_R(\bar{x}) \wedge \neg\psi_R(\bar{x}) \wedge R(\bar{x})\big),\end{aligned} \tag{2.3}$$

where $\bigvee \varnothing := \mathit{false}$. For every state $\mathcal{S}$ over Υ, $T_{(\Upsilon,\Pi)}$ maps $\mathcal{S}$ to a finite structure over $\Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$ defined as follows:

- the universe of $T_{(\Upsilon,\Pi)}(\mathcal{S})$ is that of $\mathcal{S}$,

- for every $R \in \Upsilon_{\text{mem}}$, the interpretation of R in $T_{(\Upsilon,\Pi)}(\mathcal{S})$ is $\chi_R^{\mathcal{S}}$, and
- for every $R \in \Upsilon_{\text{out}}$, the interpretation of R in $T_{(\Upsilon,\Pi)}(\mathcal{S})$ is $\varphi_R^{\mathcal{S}}$,

where $\chi_R^{\mathcal{S}}$ and $\varphi_R^{\mathcal{S}}$ denote the answer relations of the queries χ_R and φ_R on $\mathcal{S}$, respectively. This concludes the definition of $T_{(\Upsilon,\Pi)}$.

Definition 2.1.2. An *ASM (relational) transducer* T is a pair (Υ, Π) consisting of a transducer vocabulary Υ and an ASM transducer program Π over Υ. ASM-T denotes the class of ASM transducers. □

For the sake of brevity, we will from now on blur the distinction between an ASM transducer $T = (\Upsilon, \Pi)$ and the relational transducer $T_{(\Upsilon,\Pi)}$ defined by it. In particular, we denote both by T.

Runs. Let T be an ASM transducer of vocabulary Υ. A *database* $\mathcal{D}$ appropriate for T is a finite structure over Υ_{db}. An *input sequence* $\bar{I}$ appropriate for T and $\mathcal{D}$ is an infinite sequence $(\mathcal{I}_i)_{i\in\omega}$ of finite structures over Υ_{in} where each $\mathcal{I}_i$ has the same universe as $\mathcal{D}$. Suppose that $\mathcal{D}$ and $\bar{I}$ are appropriate for T (and $\mathcal{D}$, respectively). The *run* ρ of T on $\mathcal{D}$ and $\bar{I}$ is an infinite sequence $(\mathcal{S}_i)_{i\in\omega}$ of states over Υ uniquely determined by the following conditions. For every $i \in \omega$,

- $\mathcal{S}_i|_{\text{in}} = \mathcal{I}_i$,
- $\mathcal{S}_i|_{\text{db}} = \mathcal{D}$, and
- $\mathcal{S}_i|_{\text{mem,out}} = T(\mathcal{S}_{i-1})$ if $i > 0$; otherwise, all relations of $\mathcal{S}_i|_{\text{mem,out}}$ are empty (i.e., in the initial state $\mathcal{S}_0$ the memory and output are empty).

The *reduct* of ρ to some vocabulary $\Upsilon' \subseteq \Upsilon$, denoted $\rho|\Upsilon'$, is the infinite sequence $(\mathcal{S}_i|\Upsilon')_{i\in\omega}$. Again, we write $\rho|_{\text{db}}$ instead of $\rho|\Upsilon_{\text{db}}$, and $\rho|_{\text{db,mem}}$ instead of $\rho|(\Upsilon_{\text{db}} \cup \Upsilon_{\text{mem}})$, etc. The *output* and *log* produced during ρ are the infinite sequences $\rho|_{\text{out}}$ and $\rho|_{\text{log}}$, respectively. Logs have been introduced in [AVFY98] to capture the semantically significant input-output behavior of relational transducers (see the finite log validation problem in the next section).

An example of a typical, admittedly very simple application of relational transducers follows. It is inspired by an example in [AVFY98] and will serve as our running example.

Example 2.1.3. The ASM transducer T_{supp} defined below specifies the business model of a supplier whose customers can order products, are billed for them, and can take delivery of an ordered product on payment of the corresponding bill. To improve readability, we display the program of T_{supp} in a slightly relaxed syntax, using nested rules (with the obvious meaning) and rules of the form (`if` $\varphi(\bar{x}, \bar{y})$ `then` $(\neg)R(\bar{x})$) where $\{\bar{x}\} \subsetneq \text{free}(\varphi) = \{\bar{x}, \bar{y}\}$. Formally, a rule of the latter form has to be replaced with (`if` $\exists\bar{y}\varphi(\bar{x}, \bar{y})$ `then` $(\neg)R(\bar{x})$).

`transducer` T_{supp}:

`relations:`
  `input:` $Order, Pay$
  `database:` $Price, Available$
  `memory:` $PastOrder$
  `output:` $SendBill, Deliver, RejectPay, RejectOrder$
  `log:` $Pay, SendBill, Deliver$

`memory rules:`
  `if` $Order(x) \wedge Available(x) \wedge \neg PastOrder(x)$ `then`
    $PastOrder(x)$
  `if` $PastOrder(x) \wedge Pay(x, y) \wedge Price(x, y)$ `then`
    $\neg PastOrder(x)$

`output rules:`
  `if` $Order(x) \wedge Available(x)$ `then`
    `if` $\neg PastOrder(x) \wedge Price(x, y)$ `then`
      $SendBill(x, y)$
    `if` $PastOrder(x)$ `then`
      $RejectOrder(x)$
  `if` $Pay(x, y)$ `then`
    `if` $PastOrder(x) \wedge Price(x, y)$ `then`
      $Deliver(x)$
    `else`
      $RejectPay(x, y)$

The following table sketches the input and output components of a run of T_{supp}:

	$\mathcal{S}_0$	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$	$\mathcal{S}_4$	$\mathcal{S}_5$
in	$Order(a)$	$Order(b)$	$Pay(a, 5)$	$Order(a)$ $Order(b)$	$Pay(a, 5)$	
out		$SendBill(a, 5)$	$SendBill(b, 8)$	$Deliver(a)$	$SendBill(a, 5)$ $RejectOrder(b)$	$Deliver(a)$

Notice that the memory relation of T_{supp} is *not* cumulative. This enables customers to order one and the same product multiple times (in principle, infinitely often) during a session. T_{supp} provides an example of a relational transducer not definable in the Spocus transducer model of [AVFY98]. Since every Spocus transducer is a particularly simple ASM transducer, we conclude that ASM transducers are more powerful than Spocus transducers. □

Remark 2.1.4. (a) Alternatively, one could consider *finite* runs of relational transducers over possibly *expanding* domains [AVFY98]. The results in Section

2.4 should remain valid with respect to this notion of run if both the semantics of the guards of ASM transducer rules and the semantics of temporal formulas specifying properties of runs (see the next section) are adjusted appropriately.

(b) ASM transducers can be regarded as finitely initialized, deterministic ASMs with external relations. For instance, consider an ASM transducer $T = (\Upsilon, \Pi)$. We display a finitely initialized ASM $M = (\Upsilon', \mathit{initial}, \Pi')$ simulating T. Let the ASM vocabulary Υ', which now includes external symbols, be defined by $\Upsilon'_{\text{ext}} := \Upsilon_{\text{in}}$, $\Upsilon'_{\text{in}} := \Upsilon_{\text{db}}$, $\Upsilon'_{\text{stat}} := \varnothing$, $\Upsilon'_{\text{dyn}} := \Upsilon_{\text{mem}}$, and $\Upsilon'_{\text{out}} := \Upsilon_{\text{out}}$. We view the ASM transducer program Π as an ordinary ASM program with free variables. If $\bar{x}$ are the free variables of Π, then set $\Pi' = (\texttt{do-for-all}\ \bar{x} : \mathit{true}, \Pi(\bar{x}))$. Define *initial* such that it satisfies the conditions in Definition 1.1.9 and furthermore initializes every external relation as the empty relation. Now observe that for every database $\mathcal{D}$ and every input sequence $\bar{I}$ appropriate for T (and $\mathcal{D}$, respectively) the run of T on $\mathcal{D}$ and $\bar{I}$ coincides with the run of M on input $\mathcal{D}$ and external behavior $\bar{I}$. □

Computational Power of ASM Transducers

We compare the computational power of ASM transducers with the computational power of Turing machines. Since Turing machines do not interact with their environment during run-time, we focus our attention on ASM transducers that use their database as input and perform their computation without reading any further input.

Let T be an ASM transducer of vocabulary Υ where $\Upsilon_{\text{in}} = \varnothing$ and Υ_{out} contains the two boolean symbols *halt* and *accept*. Furthermore, let Q be a boolean query on $\text{Fin}(\Upsilon)$. We say that T *computes* Q if for every database $\mathcal{D}$ appropriate for T:

1. T on $\mathcal{D}$ reaches a halting state, and
2. the first halting state reached by T on $\mathcal{D}$ is accepting iff $\mathcal{D} \in Q$.

Lemma 2.1.5. *(1) A boolean query is expressible in partial fixed-point logic* (FO+PFP) *(or, equivalently, in* Datalog($\neg\neg$)*) iff it is computable by an ASM transducer. (2) On ordered databases, ASM transducers compute precisely the class of* PSPACE*-computable boolean queries.*

Proof. To (1). Consider a boolean query Q on $\text{Fin}(\Upsilon)$. For the "if" direction suppose that the ASM transducer $T = (\Upsilon', \Pi)$ computes Q. W.l.o.g., we can assume that $\Upsilon'_{\text{mem}} = \{R_1, \ldots, R_n\}$. For each $R_i \in \Upsilon'_{\text{mem}}$, define φ_{R_i} and χ_{R_i} similar to the formulas (2.1) and (2.3) in the definition of $T_{(\Upsilon,\Pi)}$. If the arity of R_i is k, then choose a k-tuple $\bar{x}_i$ of new variables and let

$$\xi_i(R_1, \ldots, R_n, \bar{x}_i) \;:=\; \big(\neg\varphi_{\mathit{halt}} \wedge \chi_{R_i}[\bar{x}/\bar{x}_i]\big) \vee \big(\varphi_{\mathit{halt}} \wedge R_i(\bar{x}_i)\big).$$

Furthermore, choose a new set symbol R_0 and a new variable x_0, and let

$$\xi_0(R_1, \ldots, R_n, x_0) \;:=\; \varphi_{accept} \wedge (x_0 = 0).$$

Since $\Upsilon'_{\text{in}} = \varnothing$, we have $\xi_0, \xi_1, \ldots \xi_n \in \text{FO}(\Upsilon \cup \Upsilon'_{\text{mem}})$. Define the simultaneous partial fixed-point sentence χ_T over Υ as follows:

$$\chi_T \;:=\; [\text{S-PFP}_{R_0 x_0,\, R_1 \bar{x}_1,\, \ldots,\, R_m \bar{x}_m}\; \xi_0(x_0), \xi_1(\bar{x}_1), \ldots, \xi_m(\bar{x}_m)](0).$$

It is not hard to verify that χ_T defines Q. Since χ_T is equivalent to a (FO+PFP) sentence over Υ (see, e.g., [EF95, Lemma 7.3.4]), Q is definable in (FO+PFP).

For the "only if" direction suppose that the sentence $\varphi \in (\text{FO+PFP})(\Upsilon)$ defines Q. We may assume that φ has the form $[\text{PFP}_{X,\bar{x}}\; \psi(X, \bar{x})](\bar{0})$, where $\psi(X, \bar{x})$ is a FO formula over $\Upsilon \cup \{X\}$ such that the fixed-point of $\psi(X, \bar{x})$ always exists (see, e.g., [EF95, Theorem 8.4.3]). The ASM transducer T_φ defined below computes the fixed-point of $\psi(X, \bar{x})$ in a memory relation X:

`transducer` T_φ:

`relations:`
  `input:` $\varnothing$
  `database:` Υ
  `memory:` X
  `output:` *halt*, *accept*

`memory rules:`
  `if` $\psi(X, \bar{x})$ `then` $X(\bar{x})$ `else` $\neg X(\bar{x})$

`output rules:`
  `if` $\forall \bar{x}(X(\bar{x}) \leftrightarrow \psi(X, \bar{x}))$ `then`
    halt
    `if` $X(\bar{0})$ `then` *accept*

By assumption on $\psi(X, \bar{x})$, T_φ halts on all databases over Υ and computes Q.

The second assertion follows from (1) and well-known results from descriptive complexity theory [Var82, AV89, EF95, Imm98]. □

2.2 Verification Problems

In this chapter, we investigate the decidability of the following three verification problems:

(1) the problem of **verifying temporal properties** of relational transducers (which concerns the correctness of relational transducers),

(2) the **finite log validation** problem (which is related to fraud detection), and

(3) the problem of **deciding log equivalence** of two relational transducers (an investigation of which is motivated by customization of relational transducers).

The first two problems are adopted from [AVFY98]. The third problem is based on a notion of equivalence stronger than the notion of equivalence considered in [AVFY98]. Intuitively, two relational transducers are *log equivalent* if they produce the same semantically significant output whenever they run on the same database and receive the same input. We think that log equivalence can provide a useful notion of equivalence for customization of relational transducers. Formal definitions of the three verification problems follow.

Verifying Temporal Properties

This problem concerns the correctness of relational transducers and thus emerges while designing transducers. To specify requirements on relational transducers we propose *first-order temporal logic* (FTL) [Eme90, AHVdB96]. FTL is a fragment of first-order branching temporal logic (FBTL), which served as specification logic for ASMs in Chapter 1. Essentially, FTL is obtained from FBTL by removing the path quantifiers **E** and **A**. (Recall in this context that relational transducers are deterministic devices.) For the reader's convenience we recall the definition of FTL below and use the opportunity to introduce the fragment UT of FTL. UT will play an important role in Section 2.4.

Definition 2.2.1. *First-order temporal logic*, denoted FTL, is obtained from FO by means of the following additional formula-formation rule:

(T) If φ and ψ are formulas, then $\mathbf{X}\varphi$, $\varphi\mathbf{U}\psi$, and $\varphi\mathbf{B}\psi$ are formulas.

The *free* and *bound variables* of FTL formulas are defined in the obvious way.

Let T denote the closure of FO under negation, disjunction, and the above rule **(T)**. The *universal closure* of T, denoted UT, is the set of FTL formulas of the form $\forall\bar{x}\varphi$ with $\varphi \in \mathrm{T}$ and $\mathrm{free}(\varphi) \subseteq \{\bar{x}\}$. □

Semantics of FTL Formulas. Consider a run $\rho = (\mathcal{S}_i)_{i\in\omega}$ of an ASM transducer of vocabulary Υ. ρ can be viewed a particularly simple computation graph C_ρ with state set $\{\mathcal{S}_i : i \in \omega\}$, transition relation $\{(\mathcal{S}_i, \mathcal{S}_{i+1}) : i \in \omega\}$, and initial state $\mathcal{S}_0$. Let φ be a FTL formula over Υ with $\mathrm{free}(\varphi) = \{\bar{x}\}$, and let $\bar{a}$ be interpretations of $\bar{x}$ chosen from the universe of $\mathcal{S}_0$. Recall the satisfaction relation $(C, \rho, \bar{a}) \models \varphi$ defined in Chapter 1 (below Definition 1.1.3). Let

$$(\rho, \bar{a}) \models \varphi \quad :\Leftrightarrow \quad (C_\rho, \rho, \bar{a}) \models \varphi.$$

The problem of verifying temporal properties of ASM transducers can now be defined as a decision problem. Consider a class C of ASM transducers and a fragment F of FTL. The *problem of verifying C-transducers against F-specifications* is defined as follows:

VERIFY(C, F) : Given an ASM transducer $T \in C$ and a sentence $\varphi \in F$ over the vocabulary of T, decide whether every run of T satisfies φ.

Example 2.2.2. Recall from Example 2.1.3 the ASM transducer T_{supp} specifying the business model of a supplier. The supplier may want to enable customers to pay an ordered product and to simultaneously reorder the very same product. We can specify this requirement on T_{supp} by means of the UT formula φ defined below. Intuitively, φ expresses that an order for a product will be rejected in the next step only if the product is currently ordered but not correctly paid.

$$\varphi \;:=\; \forall x \mathbf{G}\Big(\mathbf{X}\mathit{RejectOrder}(x) \rightarrow \big[\mathit{PastOrder}(x) \wedge \neg\exists y\big(\mathit{Pay}(x,y) \wedge \mathit{Price}(x,y)\big)\big]\Big).$$

Does T_{supp} meet the specification φ, i.e., is $(T_{\text{supp}}, \varphi) \in$ VERIFY(ASM-T, UT)? It is not hard to see that the answer is no. However, one can upgrade T_{supp} to an ASM transducer T^{+}_{supp} such that $(T^{+}_{\text{supp}}, \varphi) \in$ VERIFY (ASM-T, UT). For instance, remove the conjunct $\neg \mathit{PastOrder}(x)$ from the guard of the first memory rule of T_{supp}, and instead add $\neg \mathit{Order}(x)$ as a new conjunct to the guard of the second memory rule. Furthermore, replace the first (nested) output rule of T_{supp} with the following rule:

```
if Order(x) ∧ Available(x) then
   if ¬PastOrder(x) ∧ Price(x, y) then
      SendBill(x, y)
   if PastOrder(x) then
      if Pay(x, y) ∧ Price(x, y) then
         SendBill(x, y)
      if ¬∃y(Pay(x, y) ∧ Price(x, y)) then
         RejectOrder(x)
```

Note that one cannot avoid the introduction of a first-order quantifier in the above rule if the supplier's business model requires immediate response on any request, i.e., if it does not allow the transducer to postpone replies to a later point of time. □

Remark 2.2.3. Although FTL does not include past-tense temporal operators one can mimic such operators using additional memory relations to record the history of ongoing transactions. □

Validating Finite Logs

This problem is related to fraud detection and arises if, e.g., for efficiency reasons, the relational transducer of a supplier is allowed to run on remote customer

sites. In such a distributed scenario it can be vitally important for the supplier to be able to verify that the transactions carried out on remote sites actually conform to the own business model. That is, the supplier needs to check whether certain transactions are valid in the sense that they are the results of runs of the transducer originally distributed to customers. Since in many applications not all of the information exchanged during a transaction is really important for the supplier (like, e.g., inquiries about prices), the semantically significant input and output relations of a relational transducer are specified as *log relations* [AVFY98]. A record of a transaction is now a finite sequence of collections of log relations, where each collection contains the log relations at a particular time of the transaction.

Let T be an ASM transducer of vocabulary Υ and let $\bar{L} = (\mathcal{L}_0, \ldots, \mathcal{L}_n)$ be a finite sequence of finite structures over Υ_{log}. $\bar{L}$ is called a *finite log* of T if there exists a run ρ of T such that $\bar{L}$ is an initial segment of $\rho|_{\text{log}}$. The *finite log validation problem* for a class $C \subseteq$ ASM-T is the following decision problem:

FIN-LOG-VAL(C) : Given an ASM transducer $T \in C$ and a finite sequence $\bar{L}$ of finite structures over the log vocabulary of T, decide whether $\bar{L}$ is a finite log of T.

Example 2.2.4. Recall that *Pay*, *SendBill*, and *Deliver* are the log relations of the supplier transducer T_{supp} in Example 2.1.3. The run of T_{supp} sketched in that example witnesses that the finite sequence $\bar{L} = (\mathcal{L}_0, \ldots, \mathcal{L}_5)$ as displayed below is a finite log of T_{supp}:

	$\mathcal{L}_0$	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_3$	$\mathcal{L}_4$	$\mathcal{L}_5$
in			$Pay(a, 5)$		$Pay(a, 5)$	
out		$SendBill(a, 5)$	$SendBill(b, 8)$	$Deliver(a)$	$SendBill(a, 5)$	$Deliver(a)$

Now consider the finite sequence $\bar{L}'$ obtain from $\bar{L}$ by moving the output tuple $SendBill(a, 5)$ from $\mathcal{L}_4$ to $\mathcal{L}_3$. Is $\bar{L}'$ a finite log of T_{supp}? The answer is no, because T_{supp} can never reach a state where both $SendBill(x, y)$ and $Deliver(x)$ hold for the same product x. This is due to the fact that T_{supp} does not allow customers to pay and reorder a product simultaneously (recall Example 2.2.2). However, there exists a run of the upgraded transducer T^+_{supp} in Example 2.2.2 witnessing that $\bar{L}'$ is a finite log of T^+_{supp}. □

Deciding Log Equivalence

The main motivation for considering this problem is customization of relational transducers. To enhance competitiveness a supplier may want to allow customers to modify or upgrade the supplier's transducer for their convenience or to conform

to their own business models. This raises the question whether the customized transducers still conform to the supplier's business model.

Consider two ASM transducers T_1 and T_2 of the same vocabulary. We say that T_1 and T_2 are *log equivalent* if for every run ρ_1 of T_1 and every run ρ_2 of T_2 the following implication holds:

$$\big(\rho_1|_{\text{db,in}} = \rho_2|_{\text{db,in}}\big) \;\Rightarrow\; \big(\rho_1|_{\text{log}} = \rho_2|_{\text{log}}\big). \tag{2.4}$$

In other words, whenever T_1 and T_2 run on the same database and receive the same input, than T_1 and T_2 produce the same log. The corresponding decision problem for a class $C \subseteq$ ASM-T is defined as follows:

LOG-EQ(C) : Given two ASM transducers $T_1, T_2 \in C$ of the same vocabulary, decide whether T_1 and T_2 are log equivalent.

Example 2.2.5. Verify that T_{supp} and T^+_{supp} in Examples 2.1.3 and 2.2.2 are not log equivalent. Observe also that adding to the program of an ASM transducer new output rules which do not affect log-output relations obviously preserves log equivalence (see also [AVFY98]). For instance, a customer may propose to add to the program of T_{supp} the following output rule:

`if` $\mathit{PendingBills} \wedge \mathit{PastOrder}(x) \wedge \mathit{Price}(x,y) \wedge \neg \mathit{Pay}(x,y)$ `then`
$\mathit{Rebill}(x,y)$

where *PendingBills* is a new input relation and *Rebill* is a new output relation. If *Rebill* is not specified as a log relation, then the obtained ASM transducer is clearly log equivalence to T_{supp}. □

2.3 Natural Restrictions

Decidability and complexity of the verification problems VERIFY(C, F), FIN-LOG-VAL(C), and LOG-EQ(C) obviously depend on the choice of the class $C \subseteq$ ASM-T (and the fragment $F \subseteq$ FTL). For instance, if we set $C =$ ASM-T (and assume that F contains the formula $\mathbf{X}\mathit{accept}$), then all three problems are undecidable by Trakhtenbrot's Theorem (see, e.g., [EF95]). This leaves us with two options for how to proceed: we may

1. impose restrictions on ASM transducers, or
2. consider simplified versions of the verification problems.

The first approach was successfully pursued in [AVFY98] and led to the Spocus transducer model. Here, we follow the second approach and attempt to solve simplified versions of the verification problems for all ASM transducers. We consider two different kinds of simplification:

- The database of an ASM transducer is given.
- The maximal input flow which an ASM transducer is exposed to is a priori limited.

Each kind of simplification induces restricted variants of the verification problems, which we define next.

Providing the Database

This restriction is applicable if the database of a relational transducer changes so rarely that it becomes feasible to adjust verification to the currently valid database. As an example, consider a relational transducer T running on a notebook computer of a salesperson. The database of T may contain the catalog of a supplier and may be stored on a CD-ROM. In view of the fact that T will run on this particular database only, one can fix the database while verifying T.

We denote by VERIFY$^{\text{db}}$ the variant of VERIFY where only runs *on a given database* are considered. Formally, VERIFY$^{\text{db}}$ is defined as follows:

VERIFY$^{\text{db}}(C, F)$: Given an ASM transducer $T \in C$, a sentence $\varphi \in F$ over the vocabulary of T, *and a database* $\mathcal{D}$ *appropriate for* T, decide whether every run of T *on* $\mathcal{D}$ satisfies φ.

The corresponding variants of FIN-LOG-VAL and LOG-EQ, denoted FIN-LOG-VAL$^{\text{db}}$ and LOG-EQ$^{\text{db}}$, respectively, are defined similarly.

Lemma 2.3.1. *There exist polynomial-time reductions such that*

- VERIFY$^{\text{db}}$(ASM-T, F) *reduces to* VERIFY(ASM-T, F) *if the fragment* F *is closed under the boolean operators and contains the formula* ($\mathbf{X}$*error*)*,*
- FIN-LOG-VAL$^{\text{db}}$(ASM-T) *reduces to* FIN-LOG-VAL(ASM-T)*, and*
- LOG-EQ$^{\text{db}}$(ASM-T) *reduces to* LOG-EQ(ASM-T)*.*

Proof. We present only the first reduction; the second and third reduction are similar. Consider an instance $(T, \varphi, \mathcal{D})$ of VERIFY$^{\text{db}}$(ASM-T, F). Let Υ be the vocabulary of T, and let D be the domain of $\mathcal{D}$. W.l.o.g., we can assume that every element $a \in D$ is denoted by some constant symbol $c_a \in \Upsilon_{\text{db}}$. (If this is not the case, enrich Υ_{db} with new constant symbols and $\mathcal{D}$ with interpretations of the new symbols. This modification of $\mathcal{D}$ is clearly polynomial-time computable.) We construct an instance (T', φ') of VERIFY(ASM-T, F) such that $(T', \varphi') \in$ VERIFY(ASM-T, F) iff $(T, \varphi, \mathcal{D}) \in$ VERIFY$^{\text{db}}$(ASM-T, F).

To the definition of T' and φ'. The following FO sentence over Υ_{db} defines $\mathcal{D}$ up to isomorphism:

$$\chi_{\mathcal{D}} \;:=\; \forall x\Big(\bigvee\nolimits_{a\in D} x = c_a\Big) \wedge \Big(\bigwedge\nolimits_{\mathcal{D}\models\gamma} \gamma\Big),$$

where γ ranges in the set of atomic and negated atomic sentences over Υ_{db}. Let *error* be a new boolean output symbol. Obtain T' from T by adding the rule (`if` $\neg\chi_{\mathcal{D}}$ `then` *error*) to the program of T, and set $\varphi' = (\neg\mathbf{X}\mathit{error}) \rightarrow \varphi$. □

Remark 2.3.2. In many cases it suffices to solve FIN-LOG-VAL$^{\text{db}}$ instead of FIN-LOG-VAL. Notice that it always suffices to validate a given transaction with respect to the database that was used during the transaction. Thus, if this database is still accessible during validation time, it suffices to solve FIN-LOG-VAL$^{\text{db}}$. This especially applies to applications where transactions can be validated during transaction time. □

Of course, for applications where the database changes frequently verifying temporal properties or deciding log equivalence with respect to a fixed database is impractical or entirely useless. Here one may apply the following restriction.

Limiting the Maximal Input Flow

This restriction is motivated by the observation that the amount of input data which a relational transducer T receives from its environment during one computation step usually does not exceed a certain limit, due to physical and technical limitations of the environment. Consequently, we may focus on runs of T, where in every state, the number of tuples in every input relation is bounded by some a priori fixed natural number N (depending only on the environment of T). This motivates the following definition.

Definition 2.3.3. Let $\rho = (\mathcal{S}_i)_{i\in\omega}$ be a run of an ASM transducer of vocabulary Υ. The *maximal input flow* of ρ is the maximum of the set $\{|R^{\mathcal{S}_i}| : R \in \Upsilon_{\text{in}}, i \in \omega\}$, where $|R^{\mathcal{S}_i}|$ denotes the cardinality of the input relation $R^{\mathcal{S}_i}$ seen as a set of tuples. □

Remark 2.3.4. Alternatively, one could define the maximal input flow to be the maximum of the total number of input tuples in every state. For technical reasons we prefer the above definition. □

Let N be a natural number. We denote by VERIFY$^{\text{in} \leq N}$ the variant of VERIFY where only runs with maximal input flow $\leq N$ are considered. Formally, this problem is defined as follows:

VERIFY$^{\text{in} \leq N}(C, F)$: Given an ASM transducer $T \in C$ and a sentence $\varphi \in F$ over the vocabulary of T, decide whether every run of T *with maximal input flow* $\leq N$ satisfies φ.

The corresponding variants of FIN-LOG-VAL and LOG-EQ, denoted FIN-LOG-VAL$^{\text{in} \leq N}$ and LOG-EQ$^{\text{in} \leq N}$, respectively, are defined along the same line.

Lemma 2.3.5. *There exist polynomial-time reductions such that*

- VERIFY$^{\text{in} \leq N}$(ASM-T, F) *reduces to* VERIFY(ASM-T, F) *if the fragment F is closed under the boolean operators and contains the formula* ($\mathbf{F}$*error*),
- FIN-LOG-VAL$^{\text{in} \leq N}$(ASM-T) *reduces to* FIN-LOG-VAL(ASM-T), *and*
- LOG-EQ$^{\text{in} \leq N}$(ASM-T) *reduces to* LOG-EQ(ASM-T).

Proof. The proof is similar to the proof of Lemma 2.3.1. Again, we present only the first reduction. Consider an instance (T, φ) of VERIFY$^{\text{in} \leq N}$(ASM-T, F). We construct an instance (T', φ') of VERIFY(ASM-T, F) such that $(T', \varphi') \in$ VERIFY(ASM-T, F) iff $(T, \varphi) \in$ VERIFY$^{\text{in} \leq N}$(ASM-T, F).

To the definition of T' and φ'. Suppose that Υ is the vocabulary of T. The following FO sentence over Υ_{in} says that the current input exceed the maximal input flow N:

$$\chi_{>N} \;:=\; \bigvee\nolimits_{R \in \Upsilon_{\text{in}}} \exists \bar{x}_1, \ldots, \bar{x}_{N+1} \left(\bigwedge\nolimits_i \bar{x}_i \in R \wedge \bigwedge\nolimits_{i \neq j} \bar{x}_i \neq \bar{x}_j \right).$$

Let *error* be a new boolean output symbol. Obtain T' from T by adding the rule (`if` $\chi_{>N}$ `then` *error*) to the program of T, and set $\varphi' = (\neg \mathbf{F} \mathit{error}) \rightarrow \varphi$. □

The next lemma shows that ASM transducers whose maximal input flow is limited can also be regarded as a restricted kind of ASM transducers. Let T^* be an ASM transducer whose output vocabulary contains the boolean symbol *error*. We call a run of T^* *error-free* if *error* does not hold in any state of the run.

Lemma 2.3.6. *Fix some natural number N. Every ASM transducer T whose output vocabulary does not contain the boolean symbol error can be modified so that the error-free runs of the obtained ASM transducer are precisely the runs of T with maximal input flow $\leq N$.*

The proof of the lemma is similar to the proof of Lemma 2.3.5 and is omitted here. Equipped with the terminology established in the last two sections we can now state our main results concerning the verifiability of ASM transducers.

2.4 Verifiability Results

We first consider ASM transducers that are supposed to run on a specific database only. For any natural number m, let VERIFY$_m$ denote the restriction of VERIFY to instances where only relation symbols of arity $\leq m$ occur. The corresponding restrictions of FIN-LOG-VAL and LOG-EQ are denoted by FIN-LOG-VAL$_m$ and LOG-EQ$_m$, respectively. Recall from Definition 2.2.1 that UT denotes the set of FTL sentences of the form $\forall \bar{x} \varphi$, where φ is built from FO formulas by means of negation, disjunction, and the temporal operators $\mathbf{X}$, $\mathbf{U}$, and $\mathbf{B}$.

Theorem 2.4.1. *The following problems are* PSPACE*-complete for any* $m \geq 0$*:*

(1) $\text{VERIFY}_m^{\text{db}}(\text{ASM-T}, \text{UT})$.

(2) $\text{FIN-LOG-VAL}_m^{\text{db}}(\text{ASM-T})$.

(3) $\text{LOG-EQ}_m^{\text{db}}(\text{ASM-T})$.

In other words, if the maximal arity of the employed relations is a priori bounded, then

- *verifying temporal properties (expressible in the fragment* $\text{UT} \subseteq \text{FTL}$*),*
- *validating finite logs, and*
- *deciding log equivalence*

are PSPACE*-complete problems for ASM transducers which are supposed to run on a specific database only.*

The proof of the containment assertions of Theorem 2.4.1, i.e., the proof that problems (1)–(3) are in PSPACE, is rather technical. We postpone this proof as well as the proofs of all subsequent containment assertions until the next section. In this section, we only present proofs for the hardness of problems.

Proof. For containment of problem (1) see Corollary 2.5.16. Containment of problems (2) and (3) is then implied by the observation following the proof of Proposition 2.5.1. Hardness of problem (1): Let SAT(QBF) denote the PSPACE-complete satisfiability problem for quantified boolean formulas (see, e.g., [Pap94]). We reduce SAT(QBF) to problem (1). For every $\varphi(X_1, \ldots, X_n) \in \text{QBF}$, define $\varphi'(x_1, \ldots, x_n) \in \text{FO}$ inductively: if $\varphi = X_i$, let $\varphi' := (x_i = 1)$; if $\varphi = \neg\psi$ or $\varphi = \psi \vee \chi$, let φ' be defined in the obvious way; otherwise, if $\varphi = \exists X_i \psi$, let $\varphi' := \exists x_i \psi'$. Now consider an instance φ of SAT(QBF), i.e., an arbitrary QBF sentence. Choose some database $\mathcal{D}$ containing the two constants 0 and 1, and obtain the FO sentence φ' from φ as described above. Let the ASM transducer T_φ be defined by the program $\{\texttt{if } \varphi' \texttt{ then } \mathit{accept}\}$. Then, $\varphi \mapsto (T_\varphi, \mathbf{X}\mathit{accept}, \mathcal{D})$ is a reduction from SAT(QBF) to problem (1).

Hardness of problems (2) and (3): For every QBF sentence φ, let T_φ be defined as above. We may assume that the input vocabulary of T_φ is empty and that the log vocabulary is $\{\mathit{accept}\}$. It is now easily verified that $\varphi \mapsto (T_\varphi, (\varnothing, \{\mathit{accept}\}), \mathcal{D})$ is a reduction from SAT(QBF) to problem (2), and that $\varphi \mapsto (T_\varphi, T_{\mathit{true}}, \mathcal{D})$ is a reduction from SAT(QBF) to problem (3). □

Problems (1)–(3) in Theorem 2.4.1 are already PSPACE-hard for a fixed transducer vocabulary and a fixed database. All three problems remain in PSPACE if the arities of database and memory relations are a priori bounded, the arities of input and output relations however not (see Remark 2.5.17). If there is no fixed

upper bound on the arities of database and memory relations, then verification becomes more expensive.

Corollary 2.4.2. *The following problems are in* EXPSPACE *for any* $N \geq 0$*:*

(1') VERIFY$^{\text{db,in} \leq N}$(ASM-T, UT).

(2') FIN-LOG-VAL$^{\text{db,in} \leq N}$(ASM-T).

(3') LOG-EQ$^{\text{db,in} \leq N}$(ASM-T).

The first and the third problem are even EXPSPACE*-complete.*

Proof. For containment see again Corollary 2.5.16 and the observation following the proof of Proposition 2.5.1. Hardness of problem (1'): Let MC(FO+PFP) denote the *model checking problem* for partial fixed point logic (FO+PFP). (The model checking problem for a logic L is as follows. Given a sentence $\varphi \in L$ and a finite structure $\mathcal{A}$, both over the same vocabulary, decide whether $\mathcal{A} \models \varphi$.) It is well-known that MC(FO+PFP) is EXPSPACE-complete [Var82, Var95]. (In database theory the complexity of MC(FO+PFP) is also called *combined complexity* of the query language (FO+PFP).) We reduce MC(FO+PFP) to problem (1').

Consider an instance $(\varphi, \mathcal{D})$ of MC(FO+PFP) and let Υ be the vocabulary of φ and $\mathcal{D}$. From φ one can obtain in polynomial time an equivalent sentence of the form $[\text{PFP}_{X,\bar{x}}\, \psi(X, \bar{x})](\bar{0})$ with $\psi(X, \bar{x}) \in \text{FO}$ (see Lemma 3.3.2 and [EF95, Theorem 8.2.4]). Now follow the "only if" direction in the proof of assertion (1) of Lemma 2.1.5 to obtain an ASM transducer T_φ with empty input vocabulary, database vocabulary Υ, and output vocabulary $\{\mathit{halt}, \mathit{accept}\}$. By construction of T_φ, for every database $\mathcal{D}'$ over Υ, we have $\mathcal{D}' \models \varphi$ iff the run of T_φ on $\mathcal{D}'$ satisfies $\mathbf{F}\mathit{accept}$. (Unlike in Lemma 3.3.2, we do not care here whether T_φ holds on all inputs.) Hence, $(\varphi, \mathcal{D}) \mapsto (T_\varphi, \mathbf{F}\mathit{accept}, \mathcal{D})$ is a reduction from MC(FO+PFP) to problem (1').

Hardness of problem (3'): Let φ, $\mathcal{D}$, and T_φ be as above. We may assume that $\{\mathit{accept}\}$ is the log vocabulary of T_φ. Let T_{id} denote the ASM transducer with the empty program and the same vocabulary as T_φ. We have $\mathcal{D} \models \varphi$ iff $(T_\varphi, T_{\text{id}}, \mathcal{D})$ is a negative instance of problem (3'). This shows that $(\varphi, \mathcal{D}) \mapsto (T_\varphi, T_{\text{id}}, \mathcal{D})$ is a reduction from MC(FO+PFP) to the complement of problem (3'). Since EXPSPACE is closed under complementation, this implies hardness of problem (3'). □

As pointed out in the last section, Theorem 2.4.1 provides a sufficiently general solution of the verification problems for applications where the design and verification of ASM transducers can be adjusted to specific databases. Since it often suffices to solve FIN-LOG-VAL$^{\text{db}}$ instead of FIN-LOG-VAL (recall Remark 2.3.2), the theorem settles the finite log validation problem for many other applications as well.

Next, we turn to ASM transducers that need to be verified for *all* databases because their databases change frequently or may even change during run time (see Remark 2.4.7 below).

ASM Transducers with Input-Bounded Quantification

Unfortunately, limiting the maximal input flow alone does not suffice to obtain decidability of any of the three verification problems for the class of all ASM transducers. This is again a consequence of Trakhtenbrot's Theorem and is due to the expressive power of first-order quantification in the guards of ASM transducer rules. However, the situation chances for ASM transducers which use, instead of unbounded first-order quantification, a kind of bounded quantification specially tailored for input-driven devices like relational transducers. The main idea is to restrict the range of first-order quantifiers to the active domain of the current input. As an example, consider the following output rule taken from the ASM transducer T^+_{supp} in Example 2.2.2:

$$\begin{array}{l}\texttt{if } \mathit{Order}(x) \wedge \mathit{Available}(x) \wedge \mathit{PastOrder}(x) \\ \quad \wedge\neg\exists y(\mathit{Pay}(x,y) \wedge \mathit{Price}(x,y)) \texttt{ then} \\ \quad \mathit{RejectOrder}(x)\end{array}$$

The quantification of the variable y in the guard of this rule is 'guarded' by the input relation *Pay*, which is why the range of y can safely be restricted to the active domain of the current input. In fact, the quantification of y in the above rule is *input-bounded* in the sense of the next definition.

Definition 2.4.3. Let Υ be a transducer vocabulary. An atomic formula of the form $R(\bar{t})$ with R in Υ_{in} (resp. Υ_{mem}, Υ_{out}) is also called an *input atom* (resp. *memory atom*, *output atom*). The *input-bounded fragment of* FO, denoted FO^{I}, is obtained from FO by replacing the formula-formation rule for first-order quantification with the following rule for *input-bounded quantification*:

(IBQ) If $\bar{x}$ is a tuple of variables, α is an input atom with $\{\bar{x}\} \subseteq \text{free}(\alpha)$, and φ is a formula such that for every memory and output atom β occurring in φ, $\text{free}(\beta) \cap \{\bar{x}\} = \varnothing$, then $\exists\bar{x}(\alpha \wedge \varphi)$ and $\forall\bar{x}(\alpha \rightarrow \varphi)$ are formulas.

An *ASM transducer with input-bounded quantification* (or *ASM^I transducer* for short) is an ASM transducer in whose program all rules are guarded by FO^{I} formulas. ASM^{I}-T denotes the class of ASM^{I} transducers.

Let T^{I} denote the closure of FO^{I} under negation, disjunction, and rule **(T)** (see Definition 2.2.1). The *universal closure* of T^{I}, denoted UT^{I}, is the set of FTL formulas of the form $\forall\bar{x}\varphi$ with $\varphi \in \text{T}^{\text{I}}$ and $\text{free}(\varphi) \subseteq \{\bar{x}\}$. □

To see an example of an ASM^{I} transducer and a UT^{I} specification, recall T^+_{supp} and φ in Example 2.2.2. Also, verify that Lemma 2.3.6 remains true for ASM^{I} transducers.

Remark 2.4.4. ASM^{I} transducers and Spocus transducers are incomparable in the following sense. There exists a run of an ASM^{I} (resp. Spocus) transducer such that no Spocus (resp. ASM^{I}) transducer can produce that run. (To see this, recall Example 2.1.3 and observe that Spocus transducers may use unrestricted projections in the guards of output rules.) □

Theorem 2.4.5. *The following problems are* PSPACE*-complete for any* $N \geq 2$ *and* $m \geq 1$*:*

(4) $\text{VERIFY}_m^{\text{in} \leq N}(\text{ASM}^{\text{I}}\text{-T}, \text{UT}^{\text{I}})$.

(5) $\text{LOG-EQ}_m^{\text{in} \leq N}(\text{ASM}^{\text{I}}\text{-T})$.

In other words, if the maximal arity of the employed relations and the maximal input flow are a priori bounded, then

- *verifying temporal properties (expressible in the fragment* $\text{UT}^{\text{I}} \subseteq \text{FTL}$*), and*
- *deciding log equivalence*

are PSPACE*-complete problems for ASM*^{I} *transducers.*

Proof. For containment of problem (4) see Corollary 2.5.11. Containment of problem (5) is then implied by the observation following the proof of Proposition 2.5.1. Hardness of problem (4): We modify the reduction which implied hardness of problem (1) (see the proof of Theorem 2.4.1). Let R be a unary relation symbol. For every $\varphi(X_1, \ldots, X_n) \in \text{QBF}$, define $\varphi'(x_1, \ldots, x_n) \in \text{FO}$ inductively as in the proof of Theorem 2.4.1, except if $\varphi = \exists X_i \psi$. In that case, let $\varphi' := \exists x_i(R(x_i) \wedge \psi')$. Set $\theta = R(0) \wedge R(1) \wedge \forall x(R(x) \rightarrow (x = 0 \vee x = 1))$. Now consider an instance φ of SAT(QBF). Let the ASM^{I} transducer T_φ with input vocabulary $\{R\}$ be defined by the program $\{\texttt{if } \theta \rightarrow \varphi' \texttt{ then } \textit{accept}\}$. Verify that φ is satisfiable iff $(T_\varphi, \mathbf{X}\textit{accept})$ is a positive instance of problem (4). (It suffices to consider runs of T_φ where in the initial state the input is $\{R(0), R(1)\}$; in all other runs, T_φ accepts in the first step.) This shows that $\varphi \mapsto (T_\varphi, \mathbf{X}\textit{accept})$ is a reduction from SAT(QBF) to problem (4).

Hardness of problem (5): For every QBF sentence φ, let T_φ be defined as above. We may assume that the log vocabulary of T_φ is $\{R, \textit{accept}\}$. Then, $\varphi \mapsto (T_\varphi, T_{\textit{true}})$ is a reduction from SAT(QBF) to problem (5). □

As before, we observe an exponential blow-up of the space complexity if there is no fixed upper bound on the arities of database and memory relations.

Corollary 2.4.6. *The following problems are in* EXPSPACE *for any* $N \geq 0$*:*

(4') $\text{VERIFY}^{\text{in} \leq N}(\text{ASM}^{\text{I}}\text{-T}, \text{UT}^{\text{I}})$.

(5') LOG-EQ$^{\text{in} \leq N}$(ASM$^{\text{I}}$-T).

Both problems are even EXPSPACE*-complete if in the guards of ASM$^{\text{I}}$ transducers quantification over constants is permitted (see Corollary 2.5.13).*

Remark 2.4.7. So far, we have only considered relational transducers whose database does not change during run time. To verify an ASM$^{\text{I}}$ transducer T with an active database, i.e., a database that can be updated during a run of T, one may proceed as follows. Suppose that $R_1, \ldots, R_n$ are the active database relations of T. Modify T so that, in the first step, it copies each R_i to a new, still empty memory relation R'_i, and then, in all subsequent steps, uses R'_i instead of R_i. Every update of an R_i is now treated as input and redirected to R'_i.

There are some disadvantages to this solution, however. By definition of input-bounded quantification, it is prohibited to quantifier inside active database relations, which are now memory relations. Furthermore, if the maximal input flow is limited, then database updates may jam customer inputs. □

2.5 Proof of Containment

In this section, we prove the containment assertions of Theorems 2.4.1 and 2.4.5 and Corollaries 2.4.6 and 2.4.2. That is, we show that problems (1)–(5) are in PSPACE and that problems (1')–(5') are in EXPSPACE. We start with the observation that it suffices to consider problems (1), (1'), (4), and (4').

Proposition 2.5.1. *(i)* FIN-LOG-VAL$^{\text{db}}$(ASM-T) *is polynomial-time reducible to the complement of* VERIFY$^{\text{db}}$(ASM-T, UT). *(ii)* LOG-EQ(ASM-T) *is polynomial-time reducible to* VERIFY(ASM-T, UT)*.*

Proof. To (i). Consider an instance $(T, \bar{L}, \mathcal{D})$ of FIN-LOG-VAL$^{\text{db}}$(ASM-T). Let Υ be the vocabulary of T. W.l.o.g., we can assume that every element of $\mathcal{D}$ is denoted by some constant symbol in Υ_{db} (recall the comment the proof of Lemma 2.3.1). Let C be the set of constant symbols in Υ_{db}, and suppose that $\bar{L} = (\mathcal{L}_0, \ldots, \mathcal{L}_n)$. Below, we define a UT sentence φ such that $(T, \bar{L}, \mathcal{D}) \in$ FIN-LOG-VAL$^{\text{db}}$(ASM-T) iff $(T, \neg\varphi, \mathcal{D}) \notin$ VERIFY$^{\text{db}}$(ASM-T, UT):

$$\begin{aligned} \varphi &:= \bigwedge\nolimits_{i=0}^{n} \mathbf{X}^i \varphi_i \\ \varphi_i &:= \bigwedge\nolimits_{\mathcal{L}_i \models \gamma} \gamma, \end{aligned}$$

where γ ranges in the set of atomic and negated atomic sentences over $\Upsilon_{\log} \cup C$.

To (ii). Consider an instance (T_1, T_2) of LOG-EQ(ASM-T) and suppose that $T_1 = (\Upsilon, \Pi_1)$ and $T_2 = (\Upsilon, \Pi_2)$. W.l.o.g., we can assume that every relation symbol in Υ occurs at least once in Π_1 or Π_2. We define an ASM transducer T'

and a UT sentence φ' such that $(T', \varphi') \in$ VERIFY(ASM-T, UT) iff $(T_1, T_2) \in$ LOG-EQ(ASM-T). The idea is to let T' step-wise simulate T_1 and T_2 in parallel so that no inconsistencies between rules in Π_1 and rules in Π_2 occur. φ' can then check whether implication (2.4) in Section 2.2 holds.

To the definition of T' and φ'. Let S be the set of relation symbols occurring in Π_2 but not in $\Upsilon_{\text{db}} \cup \Upsilon_{\text{in}}$. For every $R \in S$ introduce a new relation symbol R' of the same arity and type as R. Let Υ' be Υ enriched with the new relation symbols R'. Obtain Π_2' from Π_2 by replacing every occurrence of an $R \in S$ with R'. Set $T_1' = (\Upsilon', \Pi_1)$ and $T_2' = (\Upsilon', \Pi_2')$. T_1' and T_2' are ASM transducers that have no memory and no output symbol in common but may share some symbols in $\Upsilon_{\text{db}} \cup \Upsilon_{\text{in}}$. Set $T' = (\Upsilon', \Pi')$ where $\Pi' := \Pi_1 \cup \Pi_2$, and let

$$\varphi' \;:=\; \forall \bar{x} \mathbf{G}\Big(\bigwedge\nolimits_{R \in \Upsilon_{\text{log}} \cap \Upsilon_{\text{out}}} \big(R(\bar{x}) \leftrightarrow R'(\bar{x})\big)\Big).$$

□

The proposition remains true if ASM-T and UT are replaced with ASM^{I}-T and UT^{I}, respectively. It also holds for various combinations of restrictions of VERIFY, LOG-EQ, and FIN-LOG-VAL. In particular, one can show that problems (2) and (3) reduce to problem (1), problems (2') and (3') reduce to problem (1'), problem (5) reduces to problem (4), and problem (5') reduces to problem (4'). Hence, to prove our containment assertions it suffices to consider problems (1), (1'), (4) and (4').

In the remainder of this section we will particularly be interested in the following problem:

RUN-SAT(C, F) : Given an ASM transducer $T \in C$ and a sentence $\varphi \in F$ over the vocabulary of T, decide whether there *exists* a run of T satisfying φ.

Obviously, $(T, \varphi) \in$ VERIFY(C, F) iff $(T, \neg\varphi) \notin$ RUN-SAT$(C, \neg F)$, where $\neg F$ denotes the set of negated F formulas. Recall the two fragments T and T^{I} from Definitions 2.2.1 and 2.4.3. For any complexity class K closed under complementation and polynomial-time reductions, we have

$$\begin{aligned} \text{VERIFY}(C, \text{UT}) \in K \;&\Leftrightarrow\; \text{RUN-SAT}(C, \text{T}) \in K \\ \text{VERIFY}(C, \text{UT}^{\text{I}}) \in K \;&\Leftrightarrow\; \text{RUN-SAT}(C, \text{T}^{\text{I}}) \in K. \end{aligned}$$

This observation will justify to consider RUN-SAT in place of VERIFY in the subsequent proofs of our containment assertions.

The entire construction is presented in three steps. The first step outlines the general direction of the construction in the form of a polynomial-time reduction from RUN-SAT(ASM-T, T) to the finite satisfiability problem for transitive closure logic. Refinements of this reduction in the second and third step then imply our containment assertions.

Step 1: Reduction of RUN-SAT(ASM-T, T)

We first observe an important property of runs of ASM transducers. It allows us to concentrate on *periodic runs* whose main advantage over arbitrary runs is that they are finitely representable.

Periodic Runs

Fix an ASM transducer T of vocabulary Υ.

Definition 2.5.2. A run $\rho = (\mathcal{S}_i)_{i\in\omega}$ of T is *periodic* if there exist $s, p \in \omega$, $p \geq 1$, such that $\mathcal{S}_i = \mathcal{S}_{i+p}$ for every $i \geq s$. □

Notice that T has non-periodic runs if Υ_{in} is not empty. For instance, consider a run whose input component is non-periodic. The next lemma is a generalization of an observation in [SC85]. Recall that by T we denote the set of FTL formulas built from FO formulas by means of negation, disjunction, and the temporal operators **X**, **U**, and **B**.

Lemma 2.5.3 (Periodic Run Lemma). *Let φ be a* T *sentence over Υ. If there exists a run of the ASM transducer T satisfying φ, then there also exists a periodic run of T satisfying φ.*

Proof. Suppose that $\rho = (\mathcal{S}_i)_i$ is a run of T such that $\rho \models \varphi$. It is easy to see that every run of T contains only finitely many different states. Thus, there must exist $s, p \in \omega$, $p \geq 1$ such that $\mathcal{S}_s = \mathcal{S}_{s+p}$. Define the sequence $\rho' = (\mathcal{S}'_i)_i$ inductively: if $i < s + p$, set $\mathcal{S}'_i = \mathcal{S}_i$; otherwise set $\mathcal{S}'_i = \mathcal{S}'_{i-p}$. Since T is deterministic, ρ' is obviously a periodic run of T. However, ρ' is not necessarily a model of φ because some subformulas of φ of the form $\alpha\mathbf{U}\beta$ (asserting fulfillment of β in some future state in ρ) may not be satisfied in ρ'. We show that there exist s and p such that ρ' as defined above is a model of φ. (The construction closely follows [SC85, Theorem 4.7].)

Let $\mathrm{cl}(\varphi)$ denote the set of those subformulas of φ whose occurrence is not strictly inside some FO subformula of φ. Note that all FO formulas in $\mathrm{cl}(\varphi)$ are in fact sentences and that φ can be built from these sentences by means of disjunction, conjunction, and the temporal operators **X**, **U**, and **B**. For every $i \geq 0$, let $\rho|i$ denote the infinite sequence $(\mathcal{S}_{i+j})_j$, i.e., the suffix $\mathcal{S}_i, \mathcal{S}_{i+1}, \ldots$ of ρ, and set

$$[\rho|i] \;=\; \{\theta \in \mathrm{cl}(\varphi) : \rho|i \models \theta\}.$$

Suppose that $\mathrm{cl}(\varphi)$ contains a formula $\alpha\mathbf{U}\beta$. We say that $\alpha\mathbf{U}\beta$ *holds at* $\mathcal{S}_i$ if $\alpha\mathbf{U}\beta \in [\rho|i]$. For a formula $\alpha\mathbf{U}\beta$ that holds at $\mathcal{S}_i$ we say that $\alpha\mathbf{U}\beta$ is *fulfilled before* S_j if $i < j$ and there exists $l \in \{i, \ldots, j-1\}$ such that $\beta \in [\rho|l]$.

Claim. There exist $s \geq 0$ and $p \geq 1$ such that $\mathcal{S}_s = \mathcal{S}_{s+p}$, $[\rho|s] = [\rho|s+p]$, and every formula $\alpha\mathbf{U}\beta$ that holds at $\mathcal{S}_s$ is fulfilled before $\mathcal{S}_{s+p}$.

Proof of the claim. Consider the infinite sequence $\big((\mathcal{S}_i, [\rho|i])\big)_{i\in\omega}$ of pairs. Since there are only finitely many different states $\mathcal{S}_i$ and only finitely many different sets $[\rho|i]$, there must exist a pair in the sequence which occurs infinitely often, say, at indices $s_1, s_2, s_3, \ldots$. Let $s := s_1$. Consider a formula $\alpha\mathbf{U}\beta \in [\rho|s]$. Since $\rho|s \models \alpha\mathbf{U}\beta$, there is a $k \geq s$ such that $\rho|k \models \beta$. Let k^* be the maximum of all such k, for all formulas in $[\rho|s]$ of the form $\alpha\mathbf{U}\beta$. Let s_j be the least index in the sequence $s_1, s_2, s_3, \ldots$ such that $s_j > s$ and $s_j \geq k^*$. Set $p = s_j - s$. By the choice of s and p we immediately obtain $\mathcal{S}_s = \mathcal{S}_{s+p}$ and $[\rho|s] = [\rho|s+p]$. It is straightforward to verify that every formula $\alpha\mathbf{U}\beta$ that holds at $\mathcal{S}_s$ is fulfilled before $\mathcal{S}_{s+p}$. $\Diamond$

Claim. Let s and p be as in the first claim and let ρ' be defined inductively as above. (i) If $i < s + p$, then $[\rho'|i] = [\rho|i]$. (ii) If $i \geq s + p$, then $[\rho'|i] = [\rho'|i - p]$.

Both (i) and (ii) can be proved by induction on the construction of the formulas in $\mathrm{cl}(\varphi)$. The proof of (ii) is straightforward. The proof of (i) is quite technical and requires (ii). We omit the details here. Interesting cases can be found in [SC85] (see Lemma 4.6 there). Since (by assumption) $\rho|0 \models \varphi$, (i) in the second claim implies $\rho'|0 \models \varphi$. This completes the proof of the lemma. $\square$

One-Step Semantics in Terms of FO

Definition 2.5.4. A finite or infinite sequence $(\mathcal{S}_i)_{i\in\kappa}$ of states over Υ is *consistent* if $\mathcal{S}_i|_{\mathrm{db}} = \mathcal{S}_j|_{\mathrm{db}}$ for all $i, j \in \kappa$. $\square$

We encode finite consistent sequences (presumably representing periodic runs of T) as finite structures. Suppose that σ is such a sequence, say, $\sigma = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$. Let D be the domain of the database $\mathcal{S}_0|_{\mathrm{db}}$ and set $I = \{0, \ldots, q\}$. W.l.o.g., we can assume that D and I are disjoint. Obtain Υ^+ from Υ by

1. adding to Υ two new set symbols, say, D' and I', and
2. increasing the arity of every relation symbol in $\Upsilon_{\mathrm{in}} \cup \Upsilon_{\mathrm{mem}} \cup \Upsilon_{\mathrm{out}}$ by one.

Define the finite structure $\mathcal{A}_\sigma$ over Υ^+ (encoding σ) as follows:

- the universe of $\mathcal{A}_\sigma$ is $D \cup I$,
- D' and I' are interpreted as D and I, respectively,
- every relation symbol $R \in \Upsilon_{\mathrm{db}}$ is interpreted as in $\mathcal{S}_0|_{\mathrm{db}}$, and
- every k-ary relation symbol $R \in \Upsilon_{\mathrm{in}} \cup \Upsilon_{\mathrm{mem}} \cup \Upsilon_{\mathrm{out}}$ is interpreted as a $(k+1)$-ary relation containing a tuple $(\bar{a}, i)$ iff $\bar{a} \in R^{\mathcal{S}_i}$.

To ease notation we may subsequently use one and same letter to denote both a free variable of a formula and an interpretation of that variable. The intended meaning will be clear from the context.

Proposition 2.5.5. *From the ASM transducer T one can obtain in polynomial time a* FO *formula $\chi_T(i,i')$ over Υ^+ such that for every σ as above and all $i,i' \in \{0,\dots,q\}$*

$$\mathcal{A}_\sigma \models \chi_T[i,i'] \quad \Leftrightarrow \quad T(\mathcal{S}_i) = \mathcal{S}_{i'}|_{\mathrm{mem,out}}.$$

Proof. For each $R \in \Upsilon_{\mathrm{mem}} \cup \Upsilon_{\mathrm{out}}$, define $\varphi_R(\bar{x}), \chi_R(\bar{x}) \in \mathrm{FO}(\Upsilon)$ as in Section 2.1 (see the definition of the formulas (2.1) and (2.3) there). Let $\theta_R(\bar{x}) := \chi_R(\bar{x})$ if $R \in \Upsilon_{\mathrm{mem}}$; otherwise, $\theta_R(\bar{x}) := \varphi_R(\bar{x})$. Choose two new variables i and i', and obtain $\theta_R^+(\bar{x},i) \in \mathrm{FO}(\Upsilon^+)$ from $\theta_R(\bar{x})$ as follows:

1. replace every input, memory, and output atom of the form $R'(\bar{t})$ with $R'(\bar{t},i)$, and
2. replace every FO quantifier with the corresponding D'-bounded quantifier (e.g., replace $\exists x$ with $(\exists x \in D')$, and $\forall x$ with $(\forall x \in D')$).

Finally, define $\chi_T(i,i') \in \mathrm{FO}(\Upsilon^+)$ to be

$$\textstyle\bigwedge_{R \in \Upsilon_{\mathrm{mem}} \cup \Upsilon_{\mathrm{out}}} \big[(\forall \bar{x} \in D')\big(\theta_R^+(\bar{x},i) \leftrightarrow R(\bar{x},i')\big)\big].$$

□

Reduction

Recall that by FIN-SAT(FO+TC) we denote the finite satisfiability problem for transitive closure logic. As a first step toward a proof of our containment assertions, we display a polynomial-time reduction from RUN-SAT(ASM-T, T) to FIN-SAT(FO+TC). This reduction will serve as a guideline in the second and third step of the construction.

Consider an instance (T,φ) of RUN-SAT(ASM-T, T) and let Υ be the vocabulary of T. We are going to define a sentence $\chi_{T,\varphi} \in (\mathrm{FO+TC})(\Upsilon^+)$ which has a finite model iff there exists a run of T satisfying φ. (The construction is similar to that of $\chi_{M,\varphi}$ in the proof of Theorem 1.3.3.) W.l.o.g., we can assume that φ is in negation normal form (which means that every negation in φ occurs in front of an atomic subformula). Let $\mathrm{cl}(\varphi)$ be defined as in the proof of the Periodic Run Lemma (Lemma 2.5.3), i.e, $\mathrm{cl}(\varphi)$ is the set of those subformulas of φ whose occurrence is not strictly inside some FO subformula of φ. For every $\theta \in \mathrm{cl}(\varphi)$ do the following: obtain $\theta^+(i) \in \mathrm{FO}(\Upsilon^+)$ from θ as in the proof of Proposition 2.5.5, and introduce a new boolean variable b_θ. Furthermore, if θ has the form $\alpha \mathbf{U} \beta$, then introduce an additional boolean variable m_θ different from b_θ. By $\bar{b}$ (resp. $\bar{m}$) we denote an enumeration of the boolean variables b_θ (resp. m_θ) in some random but fixed order. Let $\mathit{next} \in \mathrm{FO}(\Upsilon^+)$ with $\mathrm{free}(\mathit{next}) \subseteq \{i, i', \bar{b}, \bar{b}', \bar{m}, \bar{m}'\}$ be defined by:

$$\mathit{next}(i\bar{b}\bar{m}, i'\bar{b}'\bar{m}') \quad := \quad (i \in I') \wedge \chi_T(i,i') \wedge \textstyle\bigwedge_{\theta \in \mathrm{cl}(\varphi)} \mathit{next}_\theta,$$

where $\chi_T(i,i')$ is obtained from T according to Proposition 2.5.5, and $next_\theta$ is

$$\begin{array}{ll}
b_\theta \to \theta^+(i), & \text{if } \theta \in \text{FO} \\
b_\theta \to (b_\alpha \vee (\wedge)\, b_\beta), & \text{if } \theta = \alpha \vee (\wedge)\, \beta \\
b_\theta \to b'_\alpha, & \text{if } \theta = \mathbf{X}\alpha \\
\big(b_\theta \to (b_\beta \vee (b_\alpha \wedge b'_\theta))\big) \wedge \big(m'_\theta \to (m_\theta \vee b_\beta)\big), & \text{if } \theta = \alpha\mathbf{U}\beta \\
b_\theta \to (b_\beta \wedge (b_\alpha \vee b'_\theta)), & \text{if } \theta = \alpha\mathbf{B}\beta.
\end{array}$$

We come to the definition of $\chi_{T,\varphi}$:

$$\begin{array}{rcl}
\chi_{T,\varphi} & := & \exists i\bar{b}, i'\bar{b}', \bar{m}'\big(initial(i) \wedge run(i\bar{b}, i'\bar{b}', \bar{m}')\big) \\
initial(i) & := & \bigwedge_{R \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}} \big((\forall \bar{x} \in D')\neg R(\bar{x}, i)\big) \\
run(i\bar{b}, i'\bar{b}', \bar{m}') & := & [\text{TC}_{i\bar{b}\bar{m}, i'\bar{b}'\bar{m}'}\ next](i\bar{b}\bar{0}, i'\bar{b}'\bar{0}) \ \wedge \\
 & & [\text{TC}^S_{i\bar{b}\bar{m}, i'\bar{b}'\bar{m}'}\ next](i'\bar{b}'\bar{0}, i'\bar{b}'\bar{m}') \ \wedge \\
 & & b_\varphi \wedge \bigwedge_{\alpha\mathbf{U}\beta \in \text{cl}(\varphi)} (b'_{\alpha\mathbf{U}\beta} \to m'_{\alpha\mathbf{U}\beta}),
\end{array}$$

where TC^S denotes the strict version of the transitive closure operator TC. (TC^S is definable in (FO+TC); see formula (1.5) in the proof of Theorem 1.3.3.) It is not difficult to verify that $\chi_{T,\varphi}$ can be obtained from (T,φ) in polynomial time and that (T,φ) is a positive instance of RUN-SAT(ASM-T, T) iff $\chi_{T,\varphi}$ is a positive instance of FIN-SAT(FO+TC). (For the "only-if" direction use the Periodic Run Lemma. For the "if" direction unwind the essential part of a model of $\chi_{T,\varphi}$ to obtain a periodic run of T satisfying φ.)

Step 2: Reduction of RUN-SAT$^{\text{in} \le N}$(ASM$^{\text{I}}$-T, T$^{\text{I}}$)

We now refine the reduction in the first step to a polynomial-time reduction from RUN-SAT$^{\text{in} \le N}$(ASM$^{\text{I}}$-T, T$^{\text{I}}$) to a decidable subproblem of FIN-SAT(FO+TC). The refinement is based on the observation that for ASM$^{\text{I}}$ transducers it suffices to consider *local runs*.

Local Runs

In the following, let T be an ASM$^{\text{I}}$ transducer of vocabulary Υ.

Definition 2.5.6. For a structure $\mathcal{A}$ over Υ, we denote by $\mathcal{A}^C$ the substructure of $\mathcal{A}$ induced by the constants of $\mathcal{A}$. A consistent sequence $(\mathcal{S}_i)_{i\in\omega}$ of states over Υ is called a *local run* of T if every relation of $(\mathcal{S}_0|_{\text{mem,out}})^C$ is the empty relation, and for every $i \in \omega$, $T(\mathcal{S}_i)^C = (\mathcal{S}_{i+1}|_{\text{mem,out}})^C$. □

Lemma 2.5.7 (Local Run Lemma). *Let φ be a T^{I} sentence over Υ. If there exists a local run of the ASM^I transducer T satisfying φ, then there also exists a genuine run of T satisfying φ.*

Proof. Assume that $\sigma = (\mathcal{S}_i)_i$ is a local run of T such that $\sigma \models \varphi$. Let $\rho' = (\mathcal{S}'_i)_i$ be the run of T on the database $\mathcal{S}_0|_{\mathrm{db}}$ and the input sequence $\sigma|_{\mathrm{in}}$. We show $\rho' \models \varphi$.

Claim. If $(\mathcal{S}'_i|_{\mathrm{mem}})^C = (\mathcal{S}_i|_{\mathrm{mem}})^C$, then $T(\mathcal{S}'_i)^C = T(\mathcal{S}_i)^C$.

Proof of the claim. Let θ be a FO formula over Υ with $\mathrm{free}(\theta) = \{\bar{x}\}$. Partition $\bar{x}$ into two tuples $\bar{y}$ and $\bar{z}$ such that a variable v occurs in $\bar{y}$ iff v occurs free in some memory atom in θ. We write $\theta(\bar{y}; \bar{z})$ to indicate that the free variables of θ are partitioned in this manner. Let D be the domain of $\mathcal{S}_0|_{\mathrm{db}}$, and let C be the set of constants of $\mathcal{S}_0|_{\mathrm{db}}$. By definition, all relations in $T(\mathcal{S}'_i)$ and $T(\mathcal{S}_i)$ are defined by FO^{I} formulas over $\Upsilon - \Upsilon_{\mathrm{out}}$. It therefore suffices to show for every $\theta(\bar{y}; \bar{z}) \in \mathrm{FO}^{\mathrm{I}}(\Upsilon - \Upsilon_{\mathrm{out}})$, all interpretations $\bar{b}$ of $\bar{y}$ chosen from C, and all interpretations $\bar{c}$ of $\bar{z}$ chosen from D: $\mathcal{S}'_i \models \theta[\bar{b}; \bar{c}]$ iff $\mathcal{S}_i \models \theta[\bar{b}; \bar{c}]$. This follows from an easy induction on the construction of $\theta(\bar{y}; \bar{z})$. $\Diamond$

The following two equations hold for every $i \in \omega$: (i) $\mathcal{S}'_i|_{\mathrm{in,db}} = \mathcal{S}_i|_{\mathrm{in,db}}$, and (ii) $(\mathcal{S}'_i|_{\mathrm{mem,out}})^C = (\mathcal{S}_i|_{\mathrm{mem,out}})^C$. Equation (i) is clear by definition of ρ'. Using the first claim, one can show equation (ii) by induction on i. The next claim and our assumption $\sigma \models \varphi$ then imply $\rho' \models \varphi$.

Claim. For every $i \in \omega$, every $\theta(\bar{x}) \in \mathrm{T}^{\mathrm{I}}(\Upsilon)$ with $\mathrm{free}(\theta) = \{\bar{x}\}$, and all interpretations $\bar{a}$ of $\bar{x}$ chosen from C, $\sigma|i \models \theta[\bar{a}]$ iff $\rho'|i \models \theta[\bar{a}]$.

The proof of the claim is by induction on the construction of $\theta(\bar{x})$ simultaneously for all i. The only interesting case is $\theta(\bar{x}) \in \mathrm{FO}^{\mathrm{I}}$. Using equations (i) and (ii), one can proceed as in the proof of the first claim in this case. We omit the details. □

In favor of a succinct formulation of our next result we introduce a new kind of bounded quantification. What follows is a condensation of the first part of Section 3.1.

Witness-Bounded Quantification

A finite set of constant symbols and variables is also called a *witness set*. For a witness set W and a variable x not in W, let $(x \in W)$ abbreviate the formula $\big(\bigvee_{v \in W} x = v\big)$. Intuitively, $(x \in W)$ holds iff the interpretation of x matches the interpretation of some symbol in W.

Definition 2.5.8. The *witness-bounded fragment* of FO, denoted FO^{W}, is obtained from FO by replacing the formula-formation rule for first-order quantification with the following rule for *witness-bounded quantification*:

(WBQ) If W is a witness set, x is a variable not in W, and φ is a formula, then $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$ are formulas.

The free and bound variables of FO^{W} formulas are defined as usual. In particular, x occurs bound in $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$, whereas all variables in the witness set W occur free in these formulas. □

We view FO^{W} as a fragment of FO where formulas of the form $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$ are mere abbreviations for $\exists x(x \in W \wedge \varphi)$ and $\forall x(x \in W \rightarrow \varphi)$, respectively. It is easily verified that FO^{W} is as expressive as the quantifier-free fragment of FO (see Proposition 3.1.2). However, FO^{W} allows us to represent (certain) quantifier-free formulas exponentially more succinct (unless PSPACE = NPTIME).

One-Step Semantics in Terms of FO^{W}

We proceed toward a reduction of $\textsc{run-sat}^{\mathrm{in} \leq N}(\mathrm{ASM}^{\mathrm{I}}\text{-T}, \mathrm{T}^{\mathrm{I}})$. Fix a natural number $N \geq 1$. In the remainder of this section we tacitly assume that with every transducer vocabulary Υ there comes an arbitrary but fixed order on Υ_{in}. Let T be an $\mathrm{ASM}^{\mathrm{I}}$ transducer of vocabulary Υ and let $\sigma = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ be a consistent sequence of states over Υ with maximal input flow $\leq N$, i.e., $|R^{\mathcal{S}_i}| \leq N$ for every $R \in \Upsilon_{\mathrm{in}}$ and every $i \in \{0, \ldots, q\}$. (One may think of σ as a finite representation of a local periodic run of T.) Because N is an upper bound on the maximal input flow of σ, the input component $\mathcal{S}_i|_{\mathrm{in}}$ of each state $\mathcal{S}_i$ can be encoded as a tuple of domain elements in such a way that the length of the tuple depends only on N and Υ_{in}. Next, we fix such an encoding.

Let D be the domain of $\mathcal{S}_0|_{\mathrm{db}}$ and let $R_1, \ldots, R_n$ be an enumeration of Υ_{in} according to the order on Υ_{in}. For each R_j, let k_j denote the arity of R_j. A $N(1 + k_j)$-tuple $\bar{d}$ of elements in D is called an *encoding* of $R_j^{\mathcal{S}_i}$ if for every $l \in \{1, \ldots, N\}$ there exist $b_l \in D$ and $\bar{c}_l \in D^{k_j}$ such that $\bar{d} = (b_1\bar{c}_1, \ldots, b_N\bar{c}_N)$ and the following two conditions hold:

1. for every $\bar{a} \in R_j^{\mathcal{S}_i}$, there exists an $l \in \{1, \ldots, N\}$ such that $b_l = 0$ and $\bar{c}_l = \bar{a}$, and

2. for every $l \in \{1, \ldots, N\}$, $b_l = 0$ implies $\bar{c}_l \in R_j^{\mathcal{S}_i}$.

Set

$$L(N, \Upsilon_{\mathrm{in}}) \;=\; \sum\nolimits_{j=1}^{n} (N(1 + k_j)).$$

An $L(N, \Upsilon_{\mathrm{in}})$-tuple $\bar{e}$ of elements in D is called an *encoding* of $\mathcal{S}_i|_{\mathrm{in}}$ if for every $j \in \{1, \ldots, n\}$ there exists an encoding $\bar{d}_j$ of $R_j^{\mathcal{S}_i}$ such that $\bar{e} = (\bar{d}_1, \ldots, \bar{d}_n)$.

Proposition 2.5.9. *Recall the definitions of* Υ^+ *and* $\mathcal{A}_\sigma$ *(prior to Proposition 2.5.5) and set* $\Upsilon^* = \Upsilon^+ - \Upsilon_{\text{in}}$ *and* $\mathcal{A}^*_\sigma = \mathcal{A}_\sigma|\Upsilon^*$*. If* N *is a priori fixed, then one can obtain from the ASM*$^{\text{I}}$ *transducer* T *in polynomial time a* FO^{W} *formula* $\chi^*_T(\bar{e}, i, i')$ *over* Υ^* *such that for every* σ *as above, all* $i, i' \in \{0, \ldots, q\}$*, and every encoding* $\bar{e}$ *of* $\mathcal{S}_i|_{\text{in}}$

$$\mathcal{A}^*_\sigma \models \chi^*_T[\bar{e}, i, i'] \quad \Leftrightarrow \quad T(\mathcal{S}_i)^C = (\mathcal{S}_{i'}|_{\text{mem, out}})^C.$$

Proof. Let $\bar{e}$ be an $L(N, \Upsilon_{\text{in}})$-tuple of pairwise distinct variables such that the variables i and i' do not occur among $\bar{e}$. For each $R_j \in \Upsilon_{\text{in}}$ there exists a quantifier-free formula $\mathit{decode}_j(\bar{e}, \bar{x})$ such that for every state $\mathcal{S}$ in σ, every encoding $\bar{e}$ of $\mathcal{S}|_{\text{in}}$, and every k_j-tuple $\bar{a}$ of elements of $\mathcal{S}$,

$$\mathcal{S} \models \mathit{decode}_j[\bar{e}, \bar{a}] \quad \Leftrightarrow \quad \bar{a} \in R_j^{\mathcal{S}}.$$

The construction of $\chi^*_T(\bar{e}, i, i')$ closely follows that of $\chi_T(i, i')$ in Proposition 2.5.5. For each $R \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$, let $\theta_R(\bar{x}) \in \text{FO}^{\text{I}}(\Upsilon)$ be defined as in the proof of Proposition 2.5.5. Let C be the set of constant symbols in Υ, and let W be the set of variables occurring in $\bar{e}$. Both C and W are witness sets. W.l.o.g., we can assume that no variable in $W \cup \{i, i'\}$ occurs in any $\theta_R(\bar{x})$. Obtain $\theta^*_R(\bar{e}, \bar{x}, i) \in \text{FO}^{\text{W}}(\Upsilon^*)$ from $\theta_R(\bar{x})$ as follows:

1. replace every input atom of the form $R_j(\bar{t})$ with $\mathit{decode}_j(\bar{e}, \bar{t})$,
2. replace every memory and output atom of the form $R'(\bar{t})$ with $R'(\bar{t}, i)$, and
3. replace every FO quantifier with the corresponding W-bounded quantifier.

Define $\chi^*_T(\bar{e}, i, i') \in \text{FO}^{\text{W}}(\Upsilon^*)$ to be

$$\textstyle\bigwedge_{R \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}} \big[(\forall \bar{x} \in C)\big(\theta^*_R(\bar{e}, \bar{x}, i) \leftrightarrow R(\bar{x}, i')\big)\big].$$

□

Reduction

Let (FO^{W}+TC) denote FO^{W} augmented with the transitive closure operator TC. An occurrence of a TC operator in a (FO^{W}+TC) formula is called *positive* if the occurrence is in the scope of an even number of negations. By (FO^{W}+posTC) we denote the set of those (FO^{W}+TC) formulas in which every occurrence of a TC operator is positive.

Theorem 2.5.10. *For any* $N \geq 0$, RUN-SAT$^{\text{in} \leq N}$(ASM^{I}-T, T^{I}) *is polynomial-time reducible to* FIN-SAT(FO^{W}+posTC)*.*

Proof. Consider an instance (T, φ) of RUN-SAT$^{\text{in} \leq N}$(ASM$^{\text{I}}$-T, T$^{\text{I}}$). Let Υ be the vocabulary of T and set $\Upsilon^* = \Upsilon^+ - \Upsilon_{\text{in}}$. We are going to define a sentence $\chi^*_{T,\varphi} \in (\text{FO}^{\text{W}}\text{+posTC})(\Upsilon^*)$ which has a finite model iff there exists a run ρ of T with maximal input flow $\leq N$ such that $\rho \models \varphi$.

Recall the reduction from RUN-SAT(ASM-T, T) to FIN-SAT(FO+TC) in the first step and let cl(φ), b_θ, m_θ, $\bar{b}$, and $\bar{m}$ be as in that reduction. Notice that every FO formula θ in cl(φ) now is a FO$^{\text{I}}$ formula. For each such θ, obtain $\theta^*(\bar{e}, i) \in \text{FO}^{\text{W}}(\Upsilon^*)$ from θ as in the proof of Proposition 2.5.9. The FO formula *next* now becomes a FO$^{\text{I}}$ formula over Υ^* with free variables among $\bar{e}$, i, i', $\bar{b}, \bar{b}'$, $\bar{m}, \bar{m}'$:

$$next^*(\bar{e}, i\bar{b}\bar{m}, i'\bar{b}'\bar{m}') \;:=\; (\bar{e} \in D') \wedge (i \in I') \wedge \chi^*_T(\bar{e}, i, i') \wedge \bigwedge\nolimits_{\theta \in \text{cl}(\varphi)} next^*_\theta,$$

where $\chi^*_T(\bar{e}, i, i')$ is obtained from T according to Proposition 2.5.9, and $next^*_\theta$ is defined as $next_\theta$, except if $\theta \in \text{FO}^{\text{I}}$. In that case, $next^*_\theta := b_\theta \rightarrow \theta^*(\bar{e}, i)$. Let C be the set of constant symbols in Υ_{db}.

$$\begin{aligned}
\chi^*_{T,\varphi} \;&:=\; \exists \bar{e} i \bar{b}, \bar{e}' i' \bar{b}', \bar{m}' \big(initial^*(i) \wedge run^*(\bar{e} i \bar{b}, \bar{e}' i' \bar{b}', \bar{m}') \big) \\
initial^*(i) \;&:=\; \bigwedge\nolimits_{R \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}} \big((\forall \bar{x} \in C) \neg R(\bar{x}, i) \big) \\
run^*(\bar{e} i \bar{b}, \bar{e}' i' \bar{b}', \bar{m}') \;&:=\; [\text{TC}_{\bar{e} i \bar{b} \bar{m}, \bar{e}' i' \bar{b}' \bar{m}'}\ next^*](\bar{e} i \bar{b} \bar{0}, \bar{e}' i' \bar{b}' \bar{0}) \;\wedge \\
&\qquad [\text{TC}^S_{\bar{e} i \bar{b} \bar{m}, \bar{e}' i' \bar{b}' \bar{m}'}\ next^*](\bar{e}' i' \bar{b}' \bar{0}, \bar{e}' i' \bar{b}' \bar{m}') \;\wedge \\
&\qquad b_\varphi \wedge \bigwedge\nolimits_{\alpha \mathbf{U} \beta \in \text{cl}(\varphi)} (b'_{\alpha \mathbf{U} \beta} \rightarrow m'_{\alpha \mathbf{U} \beta})
\end{aligned}$$

Observe that $\chi^*_{T,\varphi}$ as defined above is not a (FO$^{\text{W}}$+posTC) formula. To obtain the desired (FO$^{\text{W}}$+posTC) sentence remove in the definition of $\chi^*_{T,\varphi}$ the prefix of existential quantifiers and regard all free variables of the resulting formula as new constant symbols. To complete the proof, verify that $\chi^*_{T,\varphi}$ can be obtained from (T, φ) in polynomial time and that (T, φ) is a positive instance of RUN-SAT$^{\text{in} \leq N}$(ASM$^{\text{I}}$-T, T$^{\text{I}}$) iff $\chi^*_{T,\varphi}$ is a positive instance of FIN-SAT(FO$^{\text{W}}$+posTC). (For the "only-if" direction again use the Periodic Run Lemma. For the "if" direction unwind the essential part of a model of $\chi^*_{T,\varphi}$ to obtain a local run of T satisfying φ. The Local Run Lemma then yields a genuine run of T with maximal input flow $\leq N$ which is also a model of φ.) □

Corollary 2.5.11. *For any $m, N \geq 0$, problem (4) is in* PSPACE *and problem (4') is in* EXPSPACE.

Proof. Recall that by FIN-SAT$_m$ we denote the restriction of FIN-SAT to instances where only symbols of arity $\leq m$ occur. In the next chapter, we will show that FIN-SAT$_m$(FO$^{\text{W}}$+posTC) is in PSPACE and that FIN-SAT(FO$^{\text{W}}$+posTC) is in EXPSPACE (see Corollary 3.2.8 and the proviso below Corollary 3.1.5). Together with Theorem 2.5.10 we obtain RUN-SAT$^{\text{in} \leq N}_m$(ASM$^{\text{I}}$-T, T$^{\text{I}}$) $\in$ PSPACE and

RUN-SAT$^{\text{in} \leq N}$(ASM$^{\text{I}}$-T, T$^{\text{I}}$) $\in$ EXPSPACE. (It is easy to verify that the reduction in the proof of Theorem 2.5.10 also reduces RUN-SAT$^{\text{in} \leq N}_{m}$(ASM$^{\text{I}}$-T, T$^{\text{I}}$) to FIN-SAT$_{m+1}$(FO$^{\text{W}}$+posTC).) Since both PSPACE and EXPSPACE are closed under complementation and polynomial-time reductions, our observation following the definition of RUN-SAT implies the corollary. □

The following corollary of Theorem 2.5.10 will be useful in the subsequent third step of the construction. Let FO$^{\text{IC}}$ denote the fragment of FO obtained from FO$^{\text{I}}$ (see Definition 2.4.3) by means of the additional formula-formation rule **(WBQ)** for witness-bounded quantification (see Definition 2.5.8), with the restriction that this rule can only be applied to witness sets that contain only constant symbols. Define ASM$^{\text{IC}}$-T and T$^{\text{IC}}$ as ASM$^{\text{I}}$-T and T$^{\text{I}}$ in Definition 2.4.3, except that now every occurrence of FO$^{\text{I}}$ in the definition is replaced with FO$^{\text{IC}}$.

Corollary 2.5.12. *For any $N \geq 0$,* RUN-SAT$^{\text{in} \leq N}$(ASM$^{\text{IC}}$-T, T$^{\text{IC}}$) *is polynomial-time reducible to* FIN-SAT(FO$^{\text{W}}$+posTC).

The proof of the corollary is similar to that of Theorem 2.5.10. We omit the details.

Corollary 2.5.13. *The following two problems are* EXPSPACE*-complete for any $N \geq 0$:*

(4") VERIFY$^{\text{in} \leq N}$(ASM$^{\text{IC}}$-T, UT$^{\text{IC}}$).

(5") LOG-EQ$^{\text{in} \leq N}$(ASM$^{\text{IC}}$-T).

For containment of problem (4") proceed as in the proof of Corollary 2.5.11, but now use Corollary 2.5.12 instead of Theorem 2.5.10. Containment of problem (5") then follows from the observation that the second reduction in Proposition 2.5.1 also reduces problem (5") to problem (4"). Hardness of problem (5") is proved at the end of this section (see below Corollary 2.5.16).

Step 3: Reduction of RUN-SAT$^{\text{db}}$(ASM-T, T)

We provide a reduction from RUN-SAT$^{\text{db}}$(ASM-T, T) to RUN-SAT(ASM$^{\text{IC}}$-T, T$^{\text{IC}}$). This reduction together with the reduction in Corollary 2.5.12 will imply our containment assertions concerning problems (1) and (1').

Theorem 2.5.14. RUN-SAT$^{\text{db}}$(ASM-T, T) *is polynomial-time reducible to* RUN-SAT(ASM$^{\text{IC}}$-T, T$^{\text{IC}}$).

Proof. Consider an instance $(T, \varphi, \mathcal{D})$ of RUN-SAT$^{\text{db}}$(ASM-T, T) and suppose that Υ is the vocabulary of T. W.l.o.g., we can assume that every element in $\mathcal{D}$ is denoted by a constant symbol in Υ_{db}. We construct an instance (T', φ') of

RUN-SAT($\mathrm{ASM}^{\mathrm{IC}}$-T, T^{IC}) such that there exists a run of T' satisfying φ' iff there exists a run of T on $\mathcal{D}$ satisfying φ. Let C be the set of constant symbols in Υ_{db}. Obtain T' from T by replacing in the program of T every FO quantifier with the corresponding C-bounded quantifier. T' is obviously an $\mathrm{ASM}^{\mathrm{IC}}$ transducer of vocabulary Υ. T' and T are equivalent on $\mathcal{D}$ in the sense that for every infinite sequence σ of states over Υ, σ is a run of T' on $\mathcal{D}$ iff σ is a run of T on $\mathcal{D}$. Obtain φ^C from φ by replacing every FO quantifier with the corresponding C-bounded quantifier. Define $\varphi' \in \mathrm{T}^{\mathrm{IC}}(\Upsilon)$ to be

$$\varphi^C \wedge \Big(\bigwedge_{\mathcal{D} \models \gamma} \gamma \Big) \wedge \mathbf{G}\Big(\bigwedge_{R \in \Upsilon_{\mathrm{in}}} \forall \bar{x} \big(R(\bar{x}) \rightarrow \bar{x} \in C \big) \Big),$$

where γ ranges in the set of atomic and negated atomic sentences over Υ_{db}. □

Corollary 2.5.15. *(i) For any $N \geq 0$,* RUN-SAT$^{\mathrm{db,in} \leq N}$(ASM-T, T) *is polynomial-time reducible to* FIN-SAT(FO^{W}+posTC). *(ii) For any $m \geq 0$,* RUN-SAT$^{\mathrm{db}}_m$ (ASM-T, T) *is polynomial-time reducible to* FIN-SAT$_{m+1}$(FO^{W}+posTC).

Proof. Verify that Theorem 2.5.14 remains valid if we impose an upper bound on the maximal input flow of runs. The first assertion is then implied by Corollary 2.5.12. We obtain a proof of the second assertion by composing the reductions in Theorem 2.5.14 and Corollary 2.5.12. Consider an instance $(T, \varphi, \mathcal{D})$ of RUN-SAT$^{\mathrm{db}}_m$(ASM-T, T) as in the proof of Theorem 2.5.14. Let d be the cardinality of $\mathcal{D}$. First obtain from $(T, \varphi, \mathcal{D})$ an instance (T', φ') of RUN-SAT($\mathrm{ASM}^{\mathrm{IC}}$-T, T^{IC}) according to the reduction in Theorem 2.5.14. Then, with $N := d^m$, obtain from (T', φ') an instance $\chi^*_{T',\varphi'}$ of FIN-SAT(FO^{W}+posTC) according to the reduction in Corollary 2.5.12. $\chi^*_{T',\varphi'}$ is polynomial-time computable because m is a priori fixed and $L(N, \Upsilon_{\mathrm{in}})$ is polynomially bounded in the size of $\mathcal{D}$ and Υ_{in}. □

Corollary 2.5.16. *For any $m, N \geq 0$, problem (1) is in* PSPACE *and problem (1') is in* EXPSPACE.

The proof of this corollary is similar to the proof of Corollary 2.5.11. It is omitted here. We still owe the reader a proof for the hardness of problems (4") and (5") in Corollary 2.5.13.

Proof of Corollary 2.5.13. Containment has already been discussed. For hardness it suffices to consider problem (5"). We modify the reduction that implied hardness of problem (3') in Corollary 2.4.2 and reuse some ideas of the reduction in Theorem 2.5.14. Recall from the proof of Corollary 2.4.2 that by MC(FO+PFP) we denote the model checking problem for partial fixed point logic. We reduce MC(FO+PFP) to the complement of problem (5"). Since EXPSPACE is closed under complementation this will show hardness of problem (5").

Consider an instance $(\varphi, \mathcal{D})$ of MC(FO+PFP) and suppose that Υ is the vocabulary of φ and $\mathcal{D}$. We may assume that every element of $\mathcal{D}$ is denoted by some constant symbol in Υ. Let C be the set of constant symbols in Υ. We construct an $\mathrm{ASM}^{\mathrm{IC}}$ transducer $T_{\varphi,\mathcal{D}}$ such that, if T_{id} denotes the $\mathrm{ASM}^{\mathrm{IC}}$ transducer with the empty program and the same vocabulary as $T_{\varphi,\mathcal{D}}$, then $\mathcal{D} \models \varphi$ iff $(T_{\varphi,\mathcal{D}}, T_{\mathrm{id}})$ is a negative instance of problem (5"). In a first step, obtain the ASM transducer T_φ from φ as in the proof of Corollary 2.4.2. Recall that for every database $\mathcal{D}'$ over Υ, $\mathcal{D}' \models \varphi$ iff the run of T_φ on $\mathcal{D}'$ satisfies $\mathbf{F}\mathit{accept}$. In a second step, obtain the $\mathrm{ASM}^{\mathrm{IC}}$ transducer T'_φ from T_φ by replacing in the program of T_φ every FO quantifier with the corresponding C-bounded quantifier. Suppose that Π is the program of T'_φ. Set $\chi_\mathcal{D} = \bigwedge_{\mathcal{D}\models\gamma} \gamma$, where γ ranges in the set of atomic and negated atomic sentences over Υ. Finally, let the ASM transducer $T_{\varphi,\mathcal{D}}$ be defined by the program $\{\texttt{if } \chi_\mathcal{D} \texttt{ then } \Pi\}$ and define the log vocabulary of $T_{\varphi,\mathcal{D}}$ to be $\{\mathit{accept}\}$. □

Remark 2.5.17. One can arrange the definition of $\chi^*_{T,\varphi}$ in the proof of Corollary 2.5.12 so that it becomes a sentence over $\Upsilon^+ - (\Upsilon_{\mathrm{in}} \cup \Upsilon_{\mathrm{out}})$. This shows that it suffices to impose an upper bound m on the arities of database and memory relations in order to avoid an exponential blow-up of the space complexity of problems (1'), (2'), (3'), (4"), and (5"). □

3

Logical Foundation

Consider a computational problem P and a complexity class K. In order to prove that P belongs to K it suffices to present an algorithm that solves P within the complexity bounds of K. Sometimes, however, it is easier or more convenient to reduce P to a logical decision problem in K. The advantage of this approach is that reductions to logical problems often eliminate unimportant aspects of the problems being reduced (such as syntactic sugar), thereby revealing the core of those problems. We have already seen examples of this kind of reduction in the last two chapters (recall Theorem 2.5.10 and the reductions in the proofs of Theorems 1.3.3 and 1.3.7). All these reductions had in common that they are reductions to either the *finite satisfiability problem* (FIN-SAT) or the *finite validity problem* (FIN-VAL) for some fragment of transitive-closure logic.

In this chapter, we provide the logical foundation of our containment assertions in the last two chapters by determining the complexity of FIN-SAT and FIN-VAL for the *existential fragment of transitive-closure logic* (E+TC) and the *witness-bounded fragment of positive transitive-closure logic* (FO^{W}+posTC). The latter fragment is obtained from positive transitive-closure logic (FO+posTC) [Imm87, EF95, Imm98] by replacing first-order quantification with a new kind of bounded quantification, called *witness-bounded quantification*. We first present an efficient translation from (FO+posTC) into (E+TC). This will justify to focus on (E+TC). We then prove that for (E+TC) formulas over relational vocabularies FIN-SAT and FIN-VAL are in EXPSPACE. Both problems are PSPACE-complete if there exists a fixed upper bound on the arities of relational symbols. We also show that for (E+TC) formulas with function symbols (of arity ≥ 1) FIN-SAT and FIN-VAL are undecidable.

Related Work. The previous work most closely related to our investigations in this chapter is that of Levy, Mumick, Sagiv, and Shmueli on the decidability

and complexity of various problems concerning semi-positive datalog [LMSS93]. (Note in this context that (E+TC) is as expressive as *linear* semi-positive datalog; see Corollary 3.1.6.) The proof of decidability of the satisfiability problem for semi-positive datalog in [LMSS93] is based on essential the same observation underlying our proof of decidability of FIN-SAT for (E+TC) formulas over relational vocabularies. However, it remains unclear whether the complexity bounds implied by our proof can also be obtained from the proof in [LMSS93]. Most of the decidability results in this chapter are implicitly contained in a much more sophisticated investigation of an existential fragment of second-order logic by Rosen [Ros99]. Finally, we want to mention that many useful model theoretic properties of existential least fixed-point logic (E+LFP), which subsumes (E+TC), have been observed by Blass and Gurevich [BG87] and Compton [Com93].

Outline of the Chapter. In Section 3.1, we introduce the witness-bounded fragment of positive transitive-closure logic and present an efficient translation from this fragment into (E+TC). In Section 3.2, we show that FIN-SAT and FIN-VAL are decidable for (E+TC) formulas over relational vocabularies and determine the complexity of both problems. In Section 3.3, we point out that similar decidability results can be obtained for existential least fixed-point logic (E+LFP). We conclude our investigation of (E+TC) in Section 3.4 with the observation that in the presence of function symbols neither FIN-SAT nor FIN-VAL is decidable for (E+TC) formulas.

Proviso. If not stated otherwise, Υ in this chapter denotes a finite vocabulary that may contain relation and constant symbols, but **no function symbols of arity** > 0. It is convenient to assume that Υ includes the two constant symbols 0 and 1, and to focus on structures that contain (at least) two distinct elements denoted by 0 and 1, respectively. We will do so throughout this chapter. □

3.1 A Witness-Bounded Fragment of Transitive-Closure Logic

We start with some notation. A finite set of constant symbols and variables is also called a *witness set*. For a witness set W and a variable x not in W let

$$(x \in W) \;:=\; \Big(\bigvee\nolimits_{v \in W} x = v\Big).$$

Intuitively, the formula $(x \in W)$ holds iff the interpretation of x matches the interpretation of some symbol in W. Now consider a sentence φ in the Bernays-Schönfinkel-Ramsey class (see, e.g., [BGG97]):

$$\varphi \;=\; \exists x_1 \dots x_n \forall y_1 \dots y_m \psi,$$

where ψ is a quantifier-free FO formula. W.l.o.g., we can assume that the variables $x_1, \ldots, x_n, y_1, \ldots, y_m$ are pairwise distinct. Choose n new constant symbols $c_1, \ldots, c_n$ and define the witness set W to be $\{c_1, \ldots, c_n\}$ and the witness set W' to be $\{x_1, \ldots, x_n\}$. In accordance with common notation for bounded quantification, we write $(\exists x \in W)\chi$ and $(\forall y \in W')\chi$ instead of $\exists x(x \in W \wedge \chi)$ and $\forall y(y \in W' \rightarrow \chi)$, respectively. It is not difficult to prove that φ is satisfiable iff the sentence

$$(\exists x_1 \ldots x_n \in W)(\forall y_1 \ldots y_m \in W')\psi$$

is satisfiable. This immediately follows from the observation that in a model of φ the range of every universally quantified variable y_i can safely be restricted to the set of interpretations of the variables $x_1, \ldots, x_n$. We conclude that φ has a model iff it has a finite model. Furthermore, in order to investigate the satisfiability of formulas in the Bernays-Schönfinkel-Ramsey class, it suffices to consider formulas of the above form. Such formulas are *witness-bounded* in the sense of the following definition. In this section, we introduce a fragment of transitive-closure logic whose formulas are similarly witness-bounded.

Definition 3.1.1. The *witness-bounded fragment of first-order logic*, denoted FO^{W}, is obtained from FO by replacing the formula-formation rule for first-order quantification with the following rule for *witness-bounded quantification*:

(WBQ) If W is a witness set, x is a variable not in W, and φ is a formula, then $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$ are formulas.

The free and bound variables of FO^{W} formulas are defined as usual. In particular, x occurs bound in $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$, whereas all variables in the witness set W occur free in these formulas. □

We view FO^{W} as a fragment of FO where formulas of the form $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$ are mere abbreviations for $\exists x(x \in W \wedge \varphi)$ and $\forall x(x \in W \rightarrow \varphi)$, respectively. Recall that by QF we denote the quantifier-free fragment of FO, and that $\textsc{fin-sat}_m$ denotes the restriction of FIN-SAT to formulas in which only symbols of arity at most m occur.

Proposition 3.1.2. *(1)* FO^{W} *is as expressive as* QF. *(2)* $\textsc{fin-sat}(\mathrm{FO}^{\mathrm{W}})$ *and* $\textsc{fin-sat}_m(\mathrm{FO}^{\mathrm{W}})$*, for any* $m \geq 0$*, are* PSPACE*-complete.*

Essentially the proposition says that, although adding witness-bounded quantification to QF does not increase the expressive power of QF, it allows us to represent (certain) quantifier-free formulas exponentially more succinct (unless PSPACE = NPTIME). The proof of the proposition is straightforward and omitted here. (Hardness of $\textsc{fin-sat}_m(\mathrm{FO}^{\mathrm{W}})$ follows from an easy reduction from the PSPACE-complete satisfiability problem for quantified boolean formulas.)

Definition 3.1.3. The *witness-bounded fragment of transitive-closure logic*, denoted (FO^{W}+TC), is obtained from (FO+TC) by replacing the formula-formation rule for first-order quantification with rule **(WBQ)** in Definition 3.1.1.

An occurrence of a TC operator in a (FO^{W}+TC) formula is called *positive* if the occurrence is in the scope of an even number of negations. (FO^{W}+posTC) denotes the set of those (FO^{W}+TC) formulas in which every occurrence of a TC operator is positive. □

A formula of the form $[\mathrm{TC}_{\bar{x},\bar{x}'}\,\varphi](\bar{t},\bar{t}')$ is called a *simple* TC *formula* if φ is quantifier-free and $\bar{t} = \bar{0}$ and $\bar{t}' = \bar{1}$.

Lemma 3.1.4 (Normal Form Lemma). *Every* (FO^{W}+posTC) *formula* φ *is equivalent to a simple* TC *formula. The latter formula can be obtained from* φ *in polynomial time.*

Proof. If $\varphi \in \mathrm{QF}$, then φ is equivalent to $[\mathrm{TC}_{b,b'}\,\varphi](0,1)$, where b and b' are new boolean variables. For all other cases we describe a normalization procedure that consists of four phases, each of which is polynomial-time computable.

Phase 1. Transform φ into negation normal form by moving negations as far inside as possible. In the resulting formula every occurrence of a negation symbol is in front of an atomic formula. This phase is certainly polynomial-time computable. Call the obtained formula φ_1.

Phase 2. The purpose of this phase is to transform the argument tuple of every TC subformula of φ_1 to the form $(\bar{0},\bar{1})$. (This phase is necessary for phase 4.) First verify the following equivalence:

$$[\mathrm{TC}_{\bar{x},\bar{x}'}\,\psi](\bar{t},\bar{t}') \quad\equiv\quad [\mathrm{TC}_{b\bar{z},b'\bar{z}'}\,\chi](0\bar{0},1\bar{1}), \tag{1}$$

where b and b' are boolean variables, no variable among $b, b', \bar{z}, \bar{z}'$ occurs in ψ or among $\bar{t},\bar{t}'$, and

$$\begin{aligned}\chi \;:=\; & (b = 0 \wedge b' = 1 \wedge \bar{z}' = \bar{t}) \;\vee \\ & (b = 1 \wedge \bar{z} \neq \bar{t}' \wedge b' = 1 \wedge \psi[\bar{x}/\bar{z},\bar{x}'/\bar{z}']) \;\vee \\ & (b = 1 \wedge \bar{z} = \bar{t}' \wedge b' = 1 \wedge \bar{z}' = \bar{1}).\end{aligned}$$

Now apply the following rule to φ_1 as long as possible:

- Replace an occurrence of the left-hand side of equivalence (1) with the corresponding right-hand side if $\bar{t} \neq \bar{0}$ or $\bar{t}' \neq \bar{1}$.

Since this rule can be applied to every TC subformula of φ_1 at most once, the number of rule applications is bounded by the size of φ_1. On termination every TC subformula of the resulting formula has obviously the form $[\mathrm{TC}_{\bar{x},\bar{x}'}\,\chi](\bar{0},\bar{1})$. Call the resulting formula φ_2.

Polynomial-time computability: For a formula φ, let $T(\varphi)$ denote the syntax tree of φ. We estimate the size of $T(\varphi_2)$. (For simplicity it is assumed that the nodes in $T(\varphi_2)$ representing variables or constant symbols have size 1.) Intuitively, $T(\varphi_2)$ is obtained from $T(\varphi_1)$ as follows. Every node representing a formula of the form $[\mathrm{TC}_{\bar{x},\bar{x}'}\ \psi](\bar{t},\bar{t}')$ with $\bar{t} \neq \bar{0}$ or $\bar{t}' \neq \bar{1}$ is replaced with a tree template containing slots for the subtrees of the removed TC-node. In each slot the corresponding subtree (representing ψ or one of the terms $\bar{t},\bar{t}'$) is inserted. Suppose that $\bar{t}$ and $\bar{t}'$ are k-tuples. We estimate the growth of the whole tree during one such replacement step:

- According to the definition of χ, the tree template that replaces a TC-node contains $1 + k + 2k$ slots in total: one slot for the subtree representing $\psi[\bar{x}/\bar{z},\bar{x}'/\bar{z}']$, k slots for the nodes representing $\bar{t}$, and $2k$ slots for the nodes representing $\bar{t}'$ and a set of their copies.
- The size of the tree template (with empty slots) is $\leq ak + b$ for some fixed a and b.
- The size of a subtree of the replaced TC-node does not grow.

Altogether, during one replacement step the size of the whole tree grows by at most $ak + b$ nodes. Suppose that the size of $T(\varphi_1)$ is n. Because in every replacement step we have $k \leq n$ and there are at most n replacement steps to be performed, the size of $T(\varphi_2)$ is $O(n^2)$.

We estimate the time required for one replacement step. If we traverse the current tree from top to bottom, switching to replacement mode once we found a TC-node that has to be replaced, then the total number of nodes that we have to touch is at most twice the number of nodes in $T(\varphi_2)$. Thus, the time required for one replacement step is $O(n^2)$. Since there are at most n replacement steps to be performed, the total time needed in this phase is $O(n^3)$.

Phase 3. The purpose of this phase is to remove quantifiers in φ_2. Verify the following three equivalences:

$$(\forall x \in W)\psi \quad \equiv \quad [\mathrm{TC}_{\bar{b}x,\bar{b}'x'}\ \chi](\bar{0}0,\bar{1}1), \tag{2}$$

where $W = \{v_1,\ldots,v_m\}$, $\bar{b}$ and $\bar{b}'$ are two $\lceil\log(m+1)\rceil$-tuples of boolean variables, no variable among $\bar{b},\bar{b}',x'$ occurs in ψ or W, and

$$\begin{aligned}\chi \quad := \quad & (\bar{b} = \bar{0} \wedge \bar{b}' = \mathrm{bin}(1) \wedge x' = v_1) \ \vee \\ & \Big(\psi \wedge \bigvee\nolimits_{i=1}^{m-1} (\bar{b} = \mathrm{bin}(i) \wedge \bar{b}' = \mathrm{bin}(i+1) \wedge x' = v_{i+1})\Big) \ \vee \\ & (\bar{b} = \mathrm{bin}(m) \wedge \bar{b}' = \bar{1} \wedge x' = 1).\end{aligned}$$

$$(\exists x \in W)\psi \quad \equiv \quad \exists x\big(\textstyle\bigvee_{i=1}^{m}(x = v_i) \wedge \psi\big), \tag{3}$$

where $W = \{v_1, \ldots, v_m\}$.

$$\exists x\psi \quad \equiv \quad [\mathrm{TC}_{b,x,b',x'}\ \chi](00, 11), \tag{4}$$

where b and b' are boolean variables, no variable among b, b', x' occurs in ψ, and

$$\begin{aligned} \chi \quad := \quad & (b = 0 \wedge b' = 1 \wedge \psi[x/x']) \ \vee \\ & (b = 1 \wedge b' = 1 \wedge x' = 1). \end{aligned}$$

Apply the following rule to φ_2 as long as possible:

- Replace an occurrence of the left-hand side of one of the equivalences (2)–(4) with the corresponding right-hand side.

W.l.o.g., we may assume that every application of equivalence (3) to a quantifier is immediately followed by an application of equivalence (4) to the same quantifier. Subsequently, we denote such a combined application of (3) and (4) by (3)+(4). As the above rule can be applied to every witness-bounded quantifier occurring in φ_2 at most once (where we assume that an application of (3)+(4) counts as one rule application), the number of rule applications is bounded by the size of φ_2. On termination the obtained formula does not contain any witness-bounded quantifier. Call this formula φ_3.

Polynomial-time computability: Our argument resembles that in phase 2. Suppose that the size of $T(\varphi_2)$ is n. We claim that the size of $T(\varphi_3)$ is polynomially bounded in n. Consider the two tree templates induced by (2) and (3)+(4). If $W = \{v_1, \ldots, v_m\}$, then both tree templates have $1 + m$ slots (one for ψ and m for $v_1, \ldots, v_m$) and their size (with empty slots) is $O(m)$. Since $m \leq n$, every application of (2) and (3)+(4) increases the size of the current tree by at most $O(n)$ nodes. Since there are at most n replacement steps to be performed, the size of $T(\varphi_3)$ is $O(n^2)$. As in phase 2, it follows that phase 3 requires time $O(n^3)$.

Phase 4. The purpose of this phase is to join nested TC subformulas and to remove disjunctions and conjunctions of TC subformulas in φ_3. Check that the following three equivalences hold:

$$\big[\mathrm{TC}_{\bar{x},\bar{x}'}\ [\mathrm{TC}_{\bar{y},\bar{y}'}\ \psi](\bar{s}, \bar{s}')\big](\bar{t}, \bar{t}') \quad \equiv \quad [\mathrm{TC}_{\bar{x}\bar{z}\bar{y},\bar{x}'\bar{z}'\bar{y}'}\ \chi](\bar{0}\bar{0}\bar{0}, \bar{1}\bar{1}\bar{1}), \tag{5}$$

where the variables among $\bar{x}, \bar{x}', \bar{y}, \bar{y}'$ are pairwise distinct, $\bar{s}, \bar{s}', \bar{t}, \bar{t}'$ are tuples of constants, no variable among $\bar{z}, \bar{z}'$ occurs in ψ, and

$$\begin{aligned} \chi \quad := \quad & (\bar{x} = \bar{0} \wedge \bar{z} = \bar{0} \wedge \bar{y} = \bar{0} \wedge \bar{x}' = \bar{t} \wedge \bar{y}' = \bar{s}) \ \vee \\ & (\bar{z} \neq \bar{t}' \wedge \bar{y} \neq \bar{s}' \wedge \bar{x}' = \bar{x} \wedge \bar{z}' = \bar{z} \wedge \psi[\bar{x}'/\bar{z}]) \ \vee \\ & (\bar{z} \neq \bar{t}' \wedge \bar{y} = \bar{s}' \wedge \bar{x}' = \bar{z} \wedge \bar{y}' = \bar{s}) \ \vee \\ & (\bar{z} = \bar{t}' \wedge \bar{y} = \bar{s}' \wedge \bar{x}' = \bar{z}' = \bar{y}' = \bar{1}). \end{aligned}$$

(This equivalence was taken from [Imm98].)

$$\psi_1 \vee(\wedge)\, [\mathrm{TC}_{\bar{x},\bar{x}'}\, \psi_2](\bar{0},\bar{1}) \quad \equiv \quad [\mathrm{TC}_{\bar{x},\bar{x}'}\, \psi_1 \vee(\wedge)\, \psi_2](\bar{0},\bar{1}), \tag{6}$$

where $\mathrm{free}(\psi_1) \cap \{\bar{x},\bar{x}'\} = \varnothing$.

$$[\mathrm{TC}_{\bar{x},\bar{x}'}\, \psi_1](\bar{0},\bar{1}) \vee(\wedge)\, [\mathrm{TC}_{\bar{x},\bar{x}'}\, \psi_2](\bar{0},\bar{1}) \quad \equiv \quad \big(\exists(\forall) b \in \{0,1\}\big)[\mathrm{TC}_{\bar{x},\bar{x}'}\, \chi](\bar{0},\bar{1}), \tag{7}$$

where b is a boolean variable not occurring in ψ_1 nor ψ_2, and $\chi := (b = 0 \wedge \psi_1) \vee (b \neq 0 \wedge \psi_2)$. Note that the right-hand side of the last equivalence can be replaced with an equivalent formula of the form $[\mathrm{TC}_{\bar{b}\bar{x},\bar{b}'\bar{x}'}\, \chi'](\bar{0}\bar{0},\bar{1}\bar{1})$ derivable from the right-hand side of this equivalence by means of equivalences (2)–(6). We spare the reader the details and only mention here that (i) χ' is a boolean combination of ψ_1, ψ_2, and some equations, (ii) both ψ_1 and ψ_2 occur only once in χ', and (iii) the length of $\bar{b}$ and $\bar{b}'$ can be bound by some constant.

Apply the following rule to φ_3 as long as possible:

- Replace an occurrence of the left-hand side of one of the equivalences (5)–(7) with the corresponding right-hand side if ψ (resp. ψ_1 and ψ_2) is quantifier-free. (The latter condition ensures a bottom-up/innermost-first replacement strategy).

Notice that in order to apply the above rule it might be necessary to rename variables bound by TC operators. We omit the details and observe only that every variable renaming step can be performed in time $O(n')$ where n' is the length of the output formula. (n' is polynomially bounded in the length of φ_3, as we will show.) In the following we assume that the right-hand side of equivalence (7) has already the simplified form $[\mathrm{TC}_{\bar{b}\bar{x},\bar{b}'\bar{x}'}\, \chi'](\bar{0}\bar{0},\bar{1}\bar{1})$ mentioned above.

Let n be the size of $T(\varphi_3)$. We claim that the above rule can be applied at most n times and that, after termination, the obtained formula is a simple TC formula. Consider $T(\varphi_3)$. Due to the bottom-up replacement strategy every node in $T(\varphi_3)$ is subject of a replacement step at most once. Every new node inserted during this phase will not be subject of any replacement step. This shows termination after at most n replacement steps. Call the obtained formula φ_4.

For every formula $\psi \in (\mathrm{FO}^{\mathrm{W}}{+}\mathrm{posTC})$ let the *weight* of ψ be the number of TC operators occurring in ψ. We show that the weight of φ_4 is 1. First observe that, due to our assumption that φ is not quantifier-free, the weight of φ_3 must be ≥ 1. Obviously, an application of the above rule never lowers the weight of a formula below 1, which implies that the weight of φ_4 is also ≥ 1. On the other hand, the weight of φ_4 cannot be ≥ 2, because otherwise one of the equivalences (5), (6), or (7) would be applicable. But this would contradict the definition of φ_4. (Notice that every formula occurring in this phase can be built from quantifier-free formulas by means of disjunction, conjunction, and TC operators.) Hence, the weight of φ_4 is 1. Non-applicability of equivalence (6) now implies that φ_4 is indeed a simple TC formula.

Polynomial-time computability: We claim that the size of $T(\varphi_4)$ is polynomially bounded in n. Consider the three tree templates induced by equivalences (5), (6), and (7).

- (5): Assume that $\bar{t}$ and $\bar{t}'$ are k-tuples and that $\bar{s}$ and $\bar{s}'$ are l-tuples. The tree template induced by (5) has $1 + 4k + 5l$ slots (one for ψ, $4k$ for $\bar{t}, \bar{t}'$, and $5k$ for $\bar{s}, \bar{s}'$) and its size (with empty slots) is $O(k + l)$. Due to the bottom-up strategy we know $k \leq n$. Unfortunately, there is no fixed upper bound for l because the number of variables bounded by innermost TC operators may grow during an application of (5) and (7). However, l grows by at most $2n$ during every replacement step. Thus, after i replacement steps we have $l \leq (2i + 1)n$. Since there are at most n replacement steps to be performed, we conclude that any application of (5) increases the size of the current tree by at most $O(n^2)$ nodes.

- (6): Any application of (6) does not increases the size of the current tree.

- (7): Assume that $\bar{t}, \bar{t}', \bar{s}, \bar{s}'$ are l-tuples. From (7) on can obtain tree templates with at most $1 + 1 + 6l$ slots (one for ψ_1, one for ψ_2, and $6l$ for $\bar{t}, \bar{t}', \bar{s}, \bar{s}'$). The size of each of these tree templates (with empty slots) is $O(l)$. A similar argument as for (5) shows that any application of (7) increases the size of the current tree by at most $O(n^2)$ nodes.

Altogether, the total time required in phase 4 (including necessary variable renamings) is $O(n^4)$. This concludes the proof of the Normal Form Lemma. □

Recall that by (E+TC) we denote the existential fragment of (FO+TC).

Corollary 3.1.5. (FO^{W}+posTC) *is as expressive as* (E+TC).

Proof. Verify that the normalization procedure in the proof of the preceding lemma works for (E+TC) formulas as well (equivalences (2) and (3) in phase 3 are now superfluous). Hence, given a formula in either fragment, the procedure yields an equivalent simple TC formula. Every simple TC formula is by definition a formula in both fragments. □

The proof of the last corollary also shows that there exist efficient translations from (FO+posTC) into (E+TC) and vice versa. This justifies the following proviso.

Proviso. In the remainder of this chapter we focus our attention on (E+TC), noting here that in the following every occurrence of (E+TC) is freely interchangeable with (FO^{W}+posTC). □

Let Datalog($\neg_{\mathrm{EDB}}$) denote *semi-positive datalog*, i.e., datalog with inequalities and negated EDB predicates (see, e.g., [AHV95]). A Datalog($\neg_{\mathrm{EDB}}$) program is

called *linear* if the body of every rule in the program contains at most one IDB predicate. Let LinDatalog($\neg_{\text{EDB}}$) denote the class of all linear Datalog($\neg_{\text{EDB}}$) programs.

Corollary 3.1.6. (E+TC) *is as expressive as* LinDatalog($\neg_{\text{EDB}}$).

Proof. We sketch a translation from (E+TC) into LinDatalog($\neg_{\text{EDB}}$); for a translation in the other direction the reader is referred to [Grä92, Theorem 13]. Consider a sentence $\varphi \in$ (E+TC). By the proof of Corollary 3.1.5, we may assume that φ is a simple TC formula of the form $[\text{TC}_{\bar{x},\bar{x}'}\ \psi](\bar{0},\bar{1})$. W.l.o.g., we can furthermore assume that $\psi = \bigvee_i \gamma_i$, where each γ_i is a conjunction of atomic and negated atomic formulas. Let the LinDatalog($\neg_{\text{EDB}}$) program Π_φ consist of the following rules: $T(\bar{x},\bar{x})$, $T(\bar{x},\bar{x}') \leftarrow \gamma_i[\bar{x}'/\bar{y}] \wedge T(\bar{y},\bar{x}')$ for every γ_i in ψ, and $Q \leftarrow T(\bar{0},\bar{1})$. Now verify that Π_φ with the boolean answer relation Q is equivalent to φ. □

3.2 Existential Transitive-Closure Logic

In this section, we show that for (E+TC) formulas over relational vocabularies both FIN-SAT and FIN-VAL are decidable in EXPSPACE. We consider FIN-VAL first. The following lemma is an immediate consequence of more general observations in [BG87] and [Ros99].

Lemma 3.2.1 ([BG87, Ros99]). *For every* (E+TC) *sentence φ, the class of models of φ is closed under extensions. That is, if $\mathcal{A}$ is a model of φ and $\mathcal{B}$ is an extension of $\mathcal{A}$, then $\mathcal{B}$ is also model of φ.*

Corollary 3.2.2. *Any* (E+TC) *sentence is valid iff it is finitely valid.*

Proof. The "only if" direction is trivial. For the "if" direction consider an arbitrary finitely valid (E+TC) sentence φ. Every structure $\mathcal{B}$ over the vocabulary of φ has a finite substructure $\mathcal{A}$ induced by the constants of $\mathcal{B}$. Since $\mathcal{A} \models \varphi$, Lemma 3.2.1 shows $\mathcal{B} \models \varphi$. □

Corollary 3.2.3. *(1)* FIN-VAL(E+TC) *is in* EXPSPACE. *(2) For any $m \geq 0$,* FIN-VAL$_m$(E+TC) *is* PSPACE*-complete.*

Proof. To (1). Consider an (E+TC) sentence φ and suppose that Υ is the vocabulary of φ. Let n be the number of constant symbols in Υ, and let K be the class of finite structure over Υ with cardinality $\leq n$. The proof of the last corollary shows that φ is (finitely) valid iff for every $\mathcal{A} \in K$, $\mathcal{A} \models \varphi$. Let r be the number of relation symbols in Υ, and let m be the maximal arity of these symbols. Every $\mathcal{A} \in K$ can be stored in space $O(rn^m)$. One can check $\mathcal{A} \models \varphi$ in

space polynomial in both the cardinality of $\mathcal{A}$ and the length of φ [Var82]. Since n, r, and m are bounded by the length of φ, enumerating all $\mathcal{A} \in K$ (in some standard encoding) and checking $\mathcal{A} \models \varphi$ can be done in space exponential in the length of φ. Hence, FIN-VAL(E+TC) is in EXPSPACE.

To (2). For containment of FIN-VAL$_m$(E+TC) in PSPACE observe that every $\mathcal{A} \in K$ in the above procedure can be stored in space polynomial in the length of φ if m is a priori fixed. For hardness recall that by Proposition 3.1.2, FIN-SAT$_m$(FO$^{\mathrm{W}}$) is PSPACE-hard. Since FO$^{\mathrm{W}}$ is closed under negation and PSPACE is closed under complementation, FIN-VAL$_m$(FO$^{\mathrm{W}}$) is PSPACE-hard as well. Hardness of FIN-VAL$_m$(E+TC) now follows from the Normal Form Lemma (Lemma 3.1.4). □

Next, we consider FIN-SAT. Let Υ be a relational vocabulary and let $x_1, \ldots, x_n$ be pairwise distinct variables. An *atomic type τ over Υ in $x_1, \ldots, x_n$* is a maximal satisfiable set of formulas of the form $(y_1 = y_2)$, $(y_1 \neq y_2)$, $R(y_1, \ldots, y_k)$, and $\neg R(y_1, \ldots, y_k)$ with $R \in \Upsilon$ and $y_1, \ldots, y_k \in \{x_1, \ldots, x_n\}$. Since τ is finite, the conjunction of all elements in τ is a quantifier-free formula over Υ with free variables $x_1, \ldots, x_n$. Frequently, we identify τ with this formula and write $\tau(x_1, \ldots, x_n)$ to indicate that τ is an atomic type in $x_1, \ldots, x_n$. Let $\mathrm{AT}_\Upsilon(x_1, \ldots, x_n)$ denote the set of all atomic types over Υ in $x_1, \ldots, x_n$. We note for later references that the cardinality of $\mathrm{AT}_\Upsilon(x_1, \ldots, x_n)$ is bounded by $2^{(r+1)n^{m+2}}$, where r is the number of relation symbols in Υ and m is the maximal arity of these symbols. This follows from a straightforward combinatorial argument.

Definition 3.2.4. Let Υ be a vocabulary and let l and n be two natural numbers ≥ 1. For each $i \in \{0, \ldots, l\}$, choose an n-tuple $\bar{x}_i$ of variables such that the variables among $\bar{x}_0, \ldots, \bar{x}_l$ are pairwise distinct. A *generic path over Υ of length l and width n* is a formula of the form $\bigwedge_{i=1}^{l} \tau_i(\bar{x}_{i-1}, \bar{x}_i)$ where for every $i \in \{1, \ldots, l\}$, $\tau_i(\bar{x}_{i-1}, \bar{x}_i) \in \mathrm{AT}_\Upsilon(\bar{x}_{i-1}, \bar{x}_i)$. A generic path $\bigwedge_{i=1}^{l} \tau_i(\bar{x}_{i-1}, \bar{x}_i)$ is *locally consistent* if for every $i \in \{1, \ldots, l-1\}$

$$\tau_i(\bar{x}_{i-1}, \bar{x}_i) \wedge \tau_{i+1}(\bar{x}_i, \bar{x}_{i+1})$$

is satisfiable. □

Lemma 3.2.5. *Every locally consistent generic path has a finite model.*

Proof. Consider a locally consistent generic path $\bigwedge_{i=1}^{l} \tau_i(\bar{x}_{i-1}, \bar{x}_i)$ over Υ, where each $\bar{x}_i$ is an n-tuple of variables $x_{i,1}, \ldots, x_{i,n}$. Since $\bigwedge_{i=1}^{l} \tau_i(\bar{x}_{i-1}, \bar{x}_i)$ does not contain constant symbols, we may assume that no such symbol occurs in Υ. We prove the existence of a model by induction on the length l. The case $l = 1$ is trivial because $\tau_1(\bar{x}_0, \bar{x}_1)$ is by definition satisfiable. Suppose that $l > 1$. By induction hypothesis there exists a finite structure $\mathcal{A}$ and n-tuples $\bar{a}_0, \ldots, \bar{a}_{l-1} \in$

A^n such that $\mathcal{A} \models \bigwedge_{i=1}^{l-1} \tau_i[\bar{a}_{i-1}, \bar{a}_i]$. We define a finite structure $\mathcal{B}$ such that $\mathcal{A}$ is a substructure of $\mathcal{B}$. Since $\mathcal{A} \models \bigwedge_{i=1}^{l-1} \tau_i[\bar{a}_{i-1}, \bar{a}_i]$ and $\bigwedge_{i=1}^{l-1} \tau_i(\bar{x}_{i-1}, \bar{x}_i)$ is a quantifier-free formula, we also have $\mathcal{B} \models \bigwedge_{i=1}^{l-1} \tau_i[\bar{a}_{i-1}, \bar{a}_i]$. We then define a new n-tuple $\bar{a}_l \in B^n$ such that $\mathcal{B} \models \tau_l[\bar{a}_{l-1}, \bar{a}_l]$. This will imply the lemma.

To the definition of $\mathcal{B}$. For every variable $y \in \{\bar{x}_{l-1}, \bar{x}_l\}$, let $[y] := \{z : (z = y) \in \tau_l(\bar{x}_{l-1}, \bar{x}_l)\}$. For every $[y]$ do the following: if $[y] \cap \{\bar{x}_{l-1}\} = \varnothing$, introduce a new element $a_{[y]}$ not in A; otherwise, choose some $x_{l-1,p} \in [y] \cap \{\bar{x}_{l-1}\}$ and define $a_{[y]}$ to be $a_{l-1,p}$. Define the new n-tuple $\bar{a}_l$ by setting $a_{l,p} = a_{[x_{l,p}]}$ for every $p \in \{1, \ldots, n\}$. Let $A \cup \{\bar{a}_l\}$ be the universe of $\mathcal{B}$ and let for every k-ary relation symbol $R \in \Upsilon$

$$R^{\mathcal{B}} \;:=\; R^{\mathcal{A}} \cup \{(a_{[y_1]}, \ldots, a_{[y_k]}) : R(y_1, \ldots, y_k) \in \tau_l(\bar{x}_{l-1}, \bar{x}_l)\}. \tag{3.1}$$

Now check that $\mathcal{A}$ is a substructure of $\mathcal{B}$ and that $\mathcal{B} \models \tau_l[\bar{a}_{l-1}, \bar{a}_l]$. □

Remark 3.2.6. Lemma 3.2.5 remains valid if we focus our attention on ordered structures, i.e., if we assume that Υ contains the binary relation symbol $<$ and consider only structures over Υ in which $<$ is interpreted as a linear order on the universe. □

Theorem 3.2.7. FIN-SAT(E+TC) *is decidable.*

Proof. By the Normal Form Lemma (Lemma 3.1.4) it suffices to consider simple TC sentences. Suppose that $[\mathrm{TC}_{\bar{y},\bar{y}'}\, \psi](\bar{0}, \bar{1})$ is such a sentence and that Υ is its vocabulary. Fix an enumeration $c_1, \ldots, c_k$ of the constant symbols in Υ, and set $\bar{c} = c_1, \ldots, c_k$. Introduce for each c_i two new variables z_i and z_i', and let $\varphi(\bar{y}\bar{z}, \bar{y}'\bar{z}') := \psi[\bar{c}/\bar{z}] \wedge \bigwedge_{i=1}^{k}(z_i = z_i')$. It is easy to see that $[\mathrm{TC}_{\bar{y},\bar{y}'}\, \psi](\bar{0}, \bar{1})$ is equivalent to $[\mathrm{TC}_{\bar{y}\bar{z},\bar{y}'\bar{z}'}\, \varphi](\bar{0}\bar{c}, \bar{1}\bar{c})$. For the sake of brevity, set $\bar{x} = \bar{y}\bar{z}$, $\bar{x}' = \bar{y}'\bar{z}'$, $\bar{t} = \bar{0}\bar{c}$, and $\bar{t}' = \bar{1}\bar{c}$. We show how to decide finite satisfiability of $[\mathrm{TC}_{\bar{x},\bar{x}'}\, \varphi](\bar{t}, \bar{t}')$.

Let n be the length of the tuple $\bar{x}$. Call a generic path $\bigwedge_{i=1}^{l} \tau_i(\bar{x}_{i-1}, \bar{x}_i)$ over Υ of width n *compatible with* φ *and* $(\bar{t}, \bar{t}')$ if it is locally consistent and the following $2 + l$ formulas are satisfiable:

$$\tau_1[\bar{x}_0/\bar{t}] \tag{3.2}$$

$$\tau_l[\bar{x}_l/\bar{t}'] \tag{3.3}$$

$$\tau_i[\bar{x}_{i-1}/\bar{x}][\bar{x}_i/\bar{x}'] \wedge \varphi(\bar{x}, \bar{x}'), \tag{3.4}$$

where i ranges in $\{1, \ldots, l\}$.

Claim. $[\mathrm{TC}_{\bar{x},\bar{x}'}\, \varphi](\bar{t}, \bar{t}')$ is finitely satisfiable iff there exists a generic path compatible with φ and $(\bar{t}, \bar{t}')$.

Proof of the claim. For the "only if" direction suppose that $\mathcal{A}$ is a model of $[\mathrm{TC}_{\bar{x},\bar{x}'}\,\varphi](\bar{t},\bar{t}')$. That is, there exist $\bar{a}_0,\dots,\bar{a}_l \in A^n$ such that $\bar{a}_0 = \bar{t}^{\mathcal{A}}$, $\bar{a}_l = \bar{t}'^{\mathcal{A}}$, and for every $i \in \{1,\dots,l\}$, $\mathcal{A} \models \varphi[\bar{a}_{i-1},\bar{a}_i]$. Let $\bar{x}_0,\dots,\bar{x}_l$ be as in Definition 3.2.4. For every pair $(\bar{a}_{i-1},\bar{a}_i)$ there exists (exactly one) atomic type $\tau_i(\bar{x}_{i-1},\bar{x}_i) \in \mathrm{AT}_\Upsilon(\bar{x}_{i-1},\bar{x}_i)$ such that $\mathcal{A} \models \tau_i[\bar{a}_{i-1},\bar{a}_i]$. Consider the generic path $\bigwedge_{i=1}^{l}\tau_i(\bar{x}_{i-1},\bar{x}_i)$. It is locally consistent because $\mathcal{A} \models \tau_i[\bar{a}_{i-1},\bar{a}_i] \wedge \tau_{i+1}[\bar{a}_i,\bar{a}_{i+1}]$. By the choice of the τ_i, $\mathcal{A} \models \tau_0[\bar{t}^{\mathcal{A}},\bar{a}_1]$, $\mathcal{A} \models \tau_l[\bar{a}_{l-1},\bar{t}'^{\mathcal{A}}]$, and $\mathcal{A} \models \tau_i[\bar{a}_{i-1},\bar{a}_i] \wedge \varphi[\bar{a}_{i-1},\bar{a}_i]$, which demonstrates satisfiability of formulas (3.2)–(3.4).

For the "if" direction assume that $\bigwedge_{i=1}^{l}\tau_i(\bar{x}_{i-1},\bar{x}_i)$ is a generic path compatible with φ and $(\bar{t},\bar{t}')$. By Lemma 3.2.5, there exist a finite structure $\mathcal{A}$ over $\Upsilon - \{c_1,\dots,c_k\}$ and tuples $\bar{a}_0,\dots,\bar{a}_l \in A^n$ such that $\mathcal{A} \models \bigwedge_{i=1}^{l}\tau_i[\bar{a}_{i-1},\bar{a}_i]$. We extend $\mathcal{A}$ to a structure over Υ by enriching it with interpretations for the constant symbols $c_1,\dots,c_k$ as follows: for each $p \in \{1,\dots,k\}$, set $c_p^{\mathcal{A}} = a_{0,p}$. We claim that $\bar{t}^{\mathcal{A}} = \bar{a}_0$, $\bar{t}'^{\mathcal{A}} = \bar{a}_l$, and for every $i \in \{1,\dots,l\}$, $\mathcal{A} \models \varphi[\bar{a}_{i-1},\bar{a}_i]$. This will show that $(\mathcal{A},\bar{c}^{\mathcal{A}})$ is a model of $[\mathrm{TC}_{\bar{x},\bar{x}'}\,\varphi](\bar{t},\bar{t}')$, as desired. To see $\bar{t}^{\mathcal{A}} = \bar{a}_0$, recall that $\bar{t} = \bar{0}\bar{c}$ and that formula (3.2) is satisfiable. To see $\bar{t}'^{\mathcal{A}} = \bar{a}_l$, observe that by definition of φ and satisfiability of formula (3.4), $(x_{i-1,p} = x_{i,p}) \in \tau_i(\bar{x}_{i-1},\bar{x}_i)$. This and $\mathcal{A} \models \bigwedge_{i=1}^{l}\tau_i[\bar{a}_{i-1},\bar{a}_i]$ imply $c_p^{\mathcal{A}} = a_{0,p} = a_{l,p}$. $\bar{t}'^{\mathcal{A}} = \bar{a}_l$ now follows because $\bar{t}' = \bar{1}\bar{c}$ and because formula (3.3) is satisfiable. It remains to show $\mathcal{A} \models \varphi[\bar{a}_{i-1},\bar{a}_i]$. But this is an immediate consequence of $\mathcal{A} \models \tau_i[\bar{a}_{i-1},\bar{a}_i]$ and satisfiability of formula (3.4). $\Diamond$

Claim. Let N denote the cardinality of $\mathrm{AT}_\Upsilon(\bar{x},\bar{x}')$. If there exists a generic path compatible with φ and $(\bar{t},\bar{t}')$, then there exists such a path of length at most N.

Proof of the claim. Toward a contradiction, suppose that $\bigwedge_{i=1}^{l}\tau_i(\bar{x}_{i-1},\bar{x}_i)$ is a generic path compatible with φ and $(\bar{t},\bar{t}')$ whose length l is minimal and furthermore $> N$. The idea is simple: find two occurrences of one and the same atomic type along this path and cut and paste the path between the first and second occurrence. This will yield a shorter path and thus contradict our assumption that $\bigwedge_{i=1}^{l}\tau_i(\bar{x}_{i-1},\bar{x}_i)$ is a path of minimal length.

Since $l > N$, there must exist $p,q \in \{1,\dots,l\}$, $p < q$, such that $\tau_p = \tau_q[\bar{x}_{q-1}/\bar{x}_{p-1}][\bar{x}_q/\bar{x}_p]$. Define a new generic path $\bigwedge_{i=1}^{p+(l-q)}\tau'_i(\bar{x}_{i-1},\bar{x}_i)$ of length $p+(l-q) < l$ by setting

$$\tau'_i \;=\; \begin{cases} \tau_i & \text{if } i \in \{1,\dots,p\}, \\ \tau_j[\bar{x}_{j-1}/\bar{x}_{i-1}][\bar{x}_j/\bar{x}_i] & \text{if } i \in \{p+1,\dots,p+(l-q)\}, \end{cases}$$

where $j := q+(i-p)$. We show that this path is compatible with φ and $(\bar{t},\bar{t}')$. It suffices to check whether the path is locally consistent because all other criteria for compatibility with $(\bar{t},\bar{t}')$ and φ are already satisfied by the definition of the path. We only have to check the position $i = p$, i.e., satisfiability of $\tau'_p \wedge \tau'_{p+1}$. Recall that by definition of τ'_i, $\tau'_p \wedge \tau'_{p+1} = \tau_p \wedge \tau_{q+1}[\bar{x}_q/\bar{x}_p][\bar{x}_{q+1}/\bar{x}_{p+1}]$. Since

$\tau_p = \tau_q[\bar{x}_{q-1}/\bar{x}_{p-1}][\bar{x}_q/\bar{x}_p]$ by the choice of p and q, $\tau'_p \wedge \tau'_{p+1}$ is satisfiable iff $\tau_q \wedge \tau_{q+1}$ is satisfiable. But $\tau_q \wedge \tau_{q+1}$ is satisfiable because $\bigwedge_{i=1}^{l} \tau_i(\bar{x}_{i-1}, \bar{x}_i)$ was assumed to be locally consistent. $\Diamond$

The two claims show that $[\mathrm{TC}_{\bar{x},\bar{x}'}\,\varphi](\bar{t},\bar{t}')$ is finitely satisfiable iff there exists a generic path compatible with φ and $(\bar{t},\bar{t}')$ of length $\leq N$. Since N only depends on Υ and n, one can effectively enumerate all generic paths over Υ of length $\leq N$ and width n, and check whether one of them is compatible with φ and $(\bar{t},\bar{t}')$. Checking compatibility with $(\bar{t},\bar{t}')$ and φ only involves deciding satisfiability of quantifier-free formulas, which is of course decidable. $\square$

Corollary 3.2.8. *(1)* FIN-SAT(E+TC) *is* EXPSPACE*-complete. (2) For any* $m \geq 0$*,* FIN-SAT$_m$(E+TC) *is* PSPACE*-complete.*

Proof. Containment: Below we display a non-deterministic algorithm which, on input of a simple TC sentence χ, outputs *accept* iff χ is finitely satisfiable. The algorithm is written in a self-explanatory pseudo-code and invokes the following three functions:

- *decomp*: For every simple TC sentence χ, if $[\mathrm{TC}_{\bar{x},\bar{x}'}\,\varphi](\bar{t},\bar{t}')$ is obtained from χ as in the proof of Theorem 3.2.7, then $\mathit{decomp}(\chi) = (\bar{x},\bar{x}',\varphi,\bar{t},\bar{t}')$.
- *voc*: For every formula φ, $\mathit{voc}(\varphi)$ is the vocabulary of φ.
- *sat*: For every quantifier-free formula φ, if φ is satisfiable, then $\mathit{sat}(\varphi) = \mathit{true}$; otherwise, $\mathit{sat}(\varphi) = \mathit{false}$.

```
input:
  a simple TC sentence χ
output:
  accept iff χ is finitely satisfiable
program:
  Υ := voc(χ)
  (x̄, x̄', φ, t̄, t̄') := decomp(χ)
  choose τ ∈ AT_Υ(x̄, x̄')
  if ¬(sat(τ[x̄/t̄]) ∧ sat(τ ∧ φ)) then stop
  while ¬sat(τ[x̄'/t̄']) do
    choose τ' ∈ AT_Υ(x̄, x̄')
    if ¬(sat(τ' ∧ φ) ∧ sat(τ ∧ τ'[x̄'/x̄''][x̄/x̄'])) then stop
    τ := τ'
  output accept
  stop
```

Verify that the algorithm runs in space exponential in the length of the input formula χ. Furthermore, if there is a fixed upper bound m on the arities of relation symbols occurring in χ, then the algorithm runs in space polynomial in the length of χ. Using Savitch's Theorem (see, e.g., [Pap94]) the algorithm can be turned into a deterministic procedure deciding finite satisfiability of simple TC sentences.

Hardness: Recall from the proof of Corollary 2.5.13 the reduction that implied hardness of (the complement of) problem (5"). A slight modification of this reduction also shows hardness of RUN-SAT$^{\text{in} \leq N}$(ASM$^{\text{IC}}$-T, T$^{\text{IC}}$). Corollary 2.5.12 now implies hardness of FIN-SAT(FO$^{\text{W}}$+posTC) and, by our comment following the proof of Corollary 3.1.5, hardness of FIN-SAT(E+TC). Hardness of FIN-SAT$_m$(E+TC) follows from hardness of FIN-SAT$_m$(FO$^{\text{W}}$) (recall Proposition 3.1.2) and the Normal Form Lemma. □

3.3 Existential Least Fixed-Point Logic

Existential least fixed-point logic, denoted (E+LFP), is obtained from existential first-order logic by adding a least fixed-point operator (LFP). Informally, this operator assigns to any operation definable by a positive formula the least fixed-point of the operation. (For details the reader may consult [EF95].) Lemma 3.2.1 (and thus Corollary 3.2.2) remains valid if we replace (E+TC) with (E+LFP). We obtain the following corollary, whose proof is identical to the proof of Corollary 3.2.3, except that now model checking is in EXPTIME [Var82].

Corollary 3.3.1. *(1)* FIN-VAL(E+LFP) *is in* EXPSPACE. *(2) For any* $m \geq 0$, FIN-VAL$_m$(E+LFP) *is in* EXPTIME.

In the remainder of this section, we point out that the proof of Theorem 3.2.7 can be lifted to a proof of the decidability of FIN-SAT(E+LFP). This observation is due to Erich Grädel.

A formula of the form $[\text{LFP}_{X,\bar{x}}\ \varphi](\bar{t})$ is called a *simple* LFP *formula* if φ has the form $\varphi_0 \vee (\exists \bar{y}_1, \ldots, \bar{y}_r \in X)\varphi_1$, where φ_0 and φ_1 are quantifier-free formulas which do not contain the relation symbol X, and $\bar{t} = \bar{0}$.

Lemma 3.3.2. *Every* (E+LFP) *formula* φ *is equivalent to a simple* LFP *formula. The latter formula can be obtained from* φ *in polynomial time.*

To prove this lemma translate a given (E+LFP) formula $\varphi(\bar{x})$ into a Datalog $(\neg_{\text{EDB}})$ program which is then translated back into an (E+LFP) formula (see, e.g., [EF95, Theorem 8.1.4]). In this way one can obtain a formula of the form $\exists \bar{z}[\text{LFP}_{Y,\bar{y}}\ \psi(Y, \bar{y})](\bar{x}\bar{z})$ equivalent to $\varphi(\bar{x})$, where $\psi(Y, \bar{y})$ is an existential FO formula that does not contain any variable occurring among $\bar{x}\bar{z}$. Further transformations then yield a simple LFP formula equivalent to $\varphi(\bar{x})$. We omit the details.

Remark 3.3.3. Due to Grohe every (E+LFP) formula is equivalent to a simple LFP formula of the form $[\mathrm{LFP}_{X,\bar{x}}\ \varphi_0 \vee (\exists \bar{y}, \bar{z} \in X)\varphi_1](\bar{0})$, where φ_0 and φ_1 are quantifier-free and do not contain X (see [Gro94, Theorem 3.9]). However, we do not know whether his translation is polynomial-time computable. If it is, one can set $r = 2$ throughout this section. □

Satisfiability of a simple TC sentence $[\mathrm{TC}_{\bar{x},\bar{x}'}\,\varphi](\bar{0}, \bar{1})$ is witnessed by a directed φ-path connecting $\bar{0}$ and $\bar{1}$ in some structure. One can view a generic path as a representative of a class of such paths. In the case of a simple LFP sentence $[\mathrm{LFP}_{X,\bar{x}}\ \varphi](\bar{0})$ satisfiability is witnessed by an inductive φ-proof of $\bar{0}$. Such a proof can be visualized as a derivation tree. Next, we introduce *generic trees* as representatives of classes of inductive proofs.

Let $r \geq 1$ be a natural number. A *complete r-ary tree T* is a connected directed acyclic graph (V, E) with node set $V \subseteq \{1, \ldots, r\}^*$ and edge relation $E \subseteq V \times V$ such that

- $\varepsilon \in V$ and ε is the root of T (i.e., the only node in T without predecessor),
- every node $v \in V - \{\varepsilon\}$ has precisely one predecessor,
- every node $v \in V$ is either a leaf (i.e., a node without successor) or an inner node whose successors are precisely $v1, \ldots, vr$.

Definition 3.3.4. Let Υ be a vocabulary, let $T = (V, E)$ be a complete r-ary tree, and let n be a natural number ≥ 1. For each $v \in V$, choose an n-tuple $\bar{x}_v$ of variables such that for all $v, w \in V$ the variables among $\bar{x}_v, \bar{x}_w$ are pairwise distinct. A *generic tree over Υ of form T and width n* is a formula of the form $\bigwedge_{v \in T} \tau_v(\bar{x}_v, \bar{x}_{v1}, \ldots, \bar{x}_{vr})$ where for every $v \in T$, $\tau_v(\bar{x}_v, \bar{x}_{v1}, \ldots, \bar{x}_{vr}) \in \mathrm{AT}_\Upsilon(\bar{x}_v, \bar{x}_{v1}, \ldots, \bar{x}_{vr})$. A generic tree $\bigwedge_{v \in T} \tau_v(\bar{x}_v, \bar{x}_{v1}, \ldots, \bar{x}_{vr})$ is *locally consistent* if for every inner node $v \in V$ and every $i \in \{1, \ldots, r\}$

$$\tau_v(\bar{x}_v, \bar{x}_{v1}, \ldots, \bar{x}_{vr}) \wedge \tau_{vi}(\bar{x}_{vi}, \bar{x}_{vi1}, \ldots, \bar{x}_{vir})$$

is satisfiable. □

Lemma 3.3.5. *Every locally consistent generic tree has a finite model.*

The proof of this lemma is slightly more technical than the proof of Lemma 3.2.5 but along the same line. We omit it here.

Theorem 3.3.6. FIN-SAT(E+LFP) *is decidable.*

Proof. (Sketch.) By Lemma 3.3.2 it suffices to consider simple LFP sentences. From any such sentence χ one can obtain in polynomial time an equivalent sentence χ' of the form $[\mathrm{LFP}_{X,\bar{x}}\ \varphi_0 \vee (\exists \bar{y}_1, \ldots, \bar{y}_r \in X)\varphi_1](\bar{t})$ where φ_0 and φ_1 are

quantifier-free formulas which do not contain the relation symbol X nor any constant symbol. We show how to decide finite satisfiability of χ'. Let Υ be the vocabulary of χ', and let n be the length of the tuple $\bar{x}$. Call a generic tree $\bigwedge_{v\in T} \tau_v(\bar{x}_v, \bar{x}_{v1}, \ldots, \bar{x}_{vr})$ over Υ of width n *compatible with* φ_0, φ_1, *and* $\bar{t}$ if it is locally consistent and the following formulas are satisfiable:

$$\begin{aligned}
&\tau_\varepsilon[\bar{x}_\varepsilon/\bar{t}]\\
&\tau_v[\bar{x}_v/\bar{x}] \wedge \varphi_0(\bar{x})\\
&\tau_w[\bar{x}_w/\bar{x}][\bar{x}_{w1}/\bar{y}_1], \ldots, [\bar{x}_{wr}/\bar{y}_r] \wedge \varphi_1(\bar{x}, \bar{y}_1, \ldots, \bar{y}_r),
\end{aligned}$$

where v ranges in the set of leafs of T, and w ranges in the set of inner nodes of T. Let N denote the cardinality of $\mathrm{AT}_\Upsilon(\bar{x}, \bar{y}_1, \ldots, \bar{y}_r)$. Now prove the following two claims: (1) χ' is finitely satisfiable iff there exists a generic tree compatible with φ_0, φ_1, and $\bar{t}$. (2) If there exists a generic tree compatible with φ_0, φ_1, and $\bar{t}$, then there exists such a tree of depth $\leq N$. The two claims imply that χ' is finitely satisfiable iff there exists a generic tree of depth $\leq N$ compatible with φ_0, φ_1, and $\bar{t}$. Hence, one can decide finite satisfiability of χ' by enumerate all generic trees over Υ of width n and depth $\leq N$, and checking whether one of them is compatible with φ_0, φ_1, and $\bar{t}$. □

Corollary 3.3.7. *(1)* FIN-SAT(E+LFP) *is in* 2-EXPTIME. *(2) For any* $m \geq 0$, FIN-SAT$_m$(E+LFP) *is in* EXPTIME.

Proof. We provide an alternating algorithm which, on input of a simple LFP sentence χ, outputs *accept* iff χ is finitely satisfiable. As the algorithm in the proof of Corollary 3.2.8, the algorithm below is written in a pseudo-code and employs the three functions *decomp*, *voc*, and *sat*. It is assumed that, instead of a simple TC sentence, *decomp* now takes a simple LFP sentence χ as argument and yields the components of the formula χ' obtained from χ as in the proof of Theorem 3.3.6.

```
input:
  a simple LFP sentence χ
output:
  accept iff χ is finitely satisfiable
program:
  Υ := voc(χ)
  (X̄, x̄, ȳ_1, ..., ȳ_r, φ_0, φ_1, t̄) := decomp(χ)
  ∃choose τ ∈ AT_Υ(x̄, ȳ_1, ..., ȳ_r)
  if ¬sat(τ[x̄/t̄]) then stop
  while ¬sat(τ ∧ φ_0) do
    ∀choose i ∈ {1, ..., r}
    ∃choose τ' ∈ AT_Υ(x̄, ȳ_1, ..., ȳ_r)
```

```
    if ¬(sat(τ′ ∧ φ1) ∧ sat(τ ∧ τ′[ȳ1/z̄1] ... [ȳr/z̄r][x̄/ȳi])) then stop
    τ := τ′
  output accept
  stop
```

One can check that the algorithm runs in space exponential in the length of the input formula χ. If we a priori impose an upper bound m on the arities of relation symbols occurring in χ, then it requires only polynomial space. Together with Lemma 3.3.2 we obtain FIN-SAT(E+LFP) $\in$ AEXPSPACE and FIN-SAT$_m$(E+LFP) $\in$ APSPACE. The corollary now follows from AEXPSPACE $\subseteq$ 2-EXPTIME and APSPACE $\subseteq$ EXPTIME (see, e.g., [Pap94]). □

After this short excursion to fixed-point logic we now return to transitive-closure logic.

3.4 In the Presence of Function Symbols

We conclude our investigation of (E+TC) in this section with the observation that neither FIN-SAT nor FIN-VAL is decidable for simple TC formulas over vocabularies that contain two non-nullary symbols, one of which is a function symbol. For a vocabulary Υ, we denote by FIN-SAT$_\Upsilon$ the restriction of FIN-SAT to instances over Υ. FIN-VAL$_\Upsilon$ denotes the analogous restriction of FIN-VAL.

Theorem 3.4.1. *If Υ contains two non-nullary symbols, one of which is a function symbol, then both* FIN-SAT$_\Upsilon$(E+TC) *and* FIN-VAL$_\Upsilon$(E+TC) *are undecidable.*

The proof of Theorem 3.4.1 is by reduction of two undecidable problems for deterministic finite automata with two one-way input heads. For the reader's convenience we present a formal definition of this kind of automaton below.

Definition 3.4.2. A *deterministic finite 2-head automaton* (or *2-head DFA* for short) is a quintuple $(Q, \Sigma, \delta, q_0, q_{acc})$ consisting of

- a finite set Q of states,
- an input alphabet $\Sigma = \{0, 1\}$,
- a transition function $\delta : (Q \times \Sigma_\varepsilon \times \Sigma_\varepsilon) \to (Q \times \{0, +1\} \times \{0, +1\})$, where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$,
- an initial state $q_0 \in Q$, and
- an accepting state $q_{acc} \in Q$.

Let $A = (Q, \Sigma, \delta, q_0, q_{acc})$ be a 2-head DFA. A *configuration* of A is a triple $(q, w_1, w_2) \in Q \times \Sigma^* \times \Sigma^*$ (representing the situation where A is in state q, and the first and second input head of A is placed on the first symbol of w_1 and w_2, respectively.) An *input* w appropriate for A is a finite word over Σ. On an input w, A starts its computation in the *initial configuration* (q_0, w, w). The *successor configuration* of a configuration is defined as usual. If, on an input w, A can reach a configuration (q, w_1, w_2) with $q = q_{acc}$, then A *accepts* w; otherwise, A *rejects* w. $L(A)$ denotes the set of inputs accepted by A. □

Lemma 3.4.3. *For 2-head DFAs both the emptiness problem and the totality problem are undecidable. More precisely, given a 2-head DFA A, it is undecidable whether $L(A) = \varnothing$ and it is undecidable whether $L(A) = \Sigma^*$.*

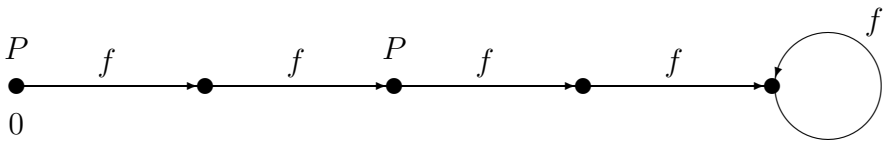
Proof. (Sketch.) Undecidability of the emptiness problem follows from a reduction from the halting problem for Turing machines to the complement of the emptiness problem. Given a Turing machine M and an input x appropriate for M, one can construct a 2-head DFA $A_{M,x}$ such that for every $w \in \Sigma^*$, $A_{M,x}$ accepts w iff w is an encoding of a halting computation of M on x. In order to verify whether w is an encoding of a computation of M, $A_{M,x}$ uses its two input heads to check whether successive segments of w encode successive configurations of a computation of M. We skip the details of the definition of $A_{M,x}$. A slight modification of $A_{M,x}$ yields also a reduction from the halting problem for Turing machines to the complement of the totality problem, and thus undecidability of the latter problem. □

Proof of Theorem 3.4.1. W.l.o.g., we can assume that Υ contains a set symbol P and a unary function symbol f. We show that (1) the complement of the emptiness problem for 2-head DFAs reduces to FIN-SAT$_\Upsilon$(E+TC), and (2) the totality problem for 2-head DFAs reduces to FIN-VAL$_\Upsilon$(E+TC). Undecidability of FIN-SAT$_\Upsilon$(E+TC) and FIN-VAL$_\Upsilon$(E+TC) then follows from Lemma 3.4.3.

To (1). For a given 2-head DFA A, we define a quantifier-free formula φ_A over Υ with $\mathrm{free}(\varphi_A) = \{\bar{b}, \bar{b}', x_1, x_1', x_2, x_2'\}$ such that for $\chi_A := \exists y_1 y_2\ [\mathrm{TC}_{\bar{b}x_1x_2,\bar{b}'x_1'x_2'}\ \varphi_A](\bar{0}00, \bar{1}y_1y_2)$

$$\chi_A \text{ is (finitely) satisfiable} \quad \Leftrightarrow \quad L(A) \neq \varnothing. \tag{3.5}$$

Before we come to the definition of φ_A, let us first provide some intuition about the construction. Suppose that $1010 \in L(A)$. The intended model of χ_A induced by 1010 is as follows:



Now consider a configuration (q, w_1, w_2) of A on input 1010. That is, q is a state of A and each w_i is a suffix of 1010. The idea is to encode (q, w_1, w_2) as a tuple $(\bar{b}, a_1, a_2)$, where $\bar{b} \in \{0,1\}^m$ encodes the state q, and a_1 and a_2 are the positions (or, elements) in the above model of χ_A where w_1 and w_2 start, respectively. We will define φ_A so that $\varphi_A[\bar{b}a_1a_2, \bar{b}'a_1'a_2']$ holds in the model iff $(\bar{b}'a_1'a_2')$ encodes the successor configuration of (q, w_1, w_2).

To the definition of φ_A. Suppose that $A = (Q, \Sigma, \delta, q_0, q_{acc})$ and that $Q = \{q_{\bar{b}} : \bar{b} \in \{0,1\}^m\}$. W.l.o.g., we can assume that $q_{\bar{0}} = q_0$ and $q_{\bar{1}} = q_{acc}$. Set $\varphi_A = \bigvee_{t\in\delta} \varphi_t$, where for every transition $t \in \delta$, φ_t is defined as follows: if $t = (q_{\bar{e}}, in_1, in_2) \to (q_{\bar{e}'}, move_1, move_2)$, then

$$\begin{aligned}
\varphi_t &:= (\bar{b} = \bar{e} \wedge \alpha_1 \wedge \alpha_2 \wedge \bar{b}' = \bar{e}' \wedge \beta_1 \wedge \beta_2) \\
\alpha_i &:= \begin{cases} x_i \neq f(x_i) \wedge Px_i & \text{if } in_i = 1 \\ x_i \neq f(x_i) \wedge \neg Px_i & \text{if } in_i = 0 \\ x_i = f(x_i) & \text{if } in_i = \varepsilon \end{cases} \\
\beta_i &:= \begin{cases} x_i' = f(x_i) & \text{if } move_i = +1 \\ x_i' = x_i & \text{if } move_i = 0. \end{cases}
\end{aligned}$$

It is not difficult show that χ_M satisfies equivalence (3.5).

To (2). For a given 2-head DFA A, we define a sentence $\chi_A' \in (\mathrm{E{+}TC})(\Upsilon)$ such that

$$\chi_A' \text{ is (finitely) valid} \quad \Leftrightarrow \quad L(A) = \Sigma^*. \tag{3.6}$$

First verify that χ_A in the proof of assertion (1) may not satisfy the "if" direction of equivalence (3.6). This is because there may exist some $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ in which the f-path starting at 0 runs into a cycle of length > 1 such that the resulting infinite f-path represents a non-accepting, infinite computation of A. In that case there may not exist a φ_A-path from $(\bar{0}00)$ to some $(\bar{1}a_1a_2)$, although $L(A) = \Sigma^*$.

We can circumvent this problem by adding to χ_A a formula that holds in all finite structures where the f-path starting at 0 leads to a cycle of length > 1. For instance, check that

$$\begin{aligned}
\chi_A' := \chi_A \vee \exists y\big(f(y) \neq y \wedge [\mathrm{TC}_{x,x'}\, x' = f(x)](0, y) \wedge \\
[\mathrm{TC}_{x,x'}\, x' = f(x)](f(y), y)\big)
\end{aligned}$$

satisfies equivalence (3.6). □

Note that by the Normal Form Lemma (Lemma 3.1.4) the sentences χ_A and χ_A' in the proof of Theorem 3.4.1 are equivalent to simple TC sentences. Hence, even for simple TC sentences over vocabularies that contain non-nullary function symbols finite satisfiability and finite validity are undecidable properties.

4

Discussion

In this first part of the thesis, we have investigated the automatic verifiability of ASMs. As a first step toward such an investigation, we have defined a decision problem, called the *verification problem for ASMs*, which concerns the question of whether a given ASM satisfies a given property (expressible in the temporal logic FBTL) on all inputs. Decidability of this problem for a class C of ASMs and a fragment F of FBTL implies that all ASMs in C can be verified automatically against specifications definable in F. We have then studied the decidability and complexity of the verification problem for several classes of restricted ASMs: in Chapter 1 for the class of *sequential nullary ASMs* and various extensions thereof, and in Chapter 2 for the class of *ASM relational transducers*.

Regarding the automatic verifiability of sequential nullary ASMs the results are twofold. On the one hand, we have proved that sequential nullary ASMs with relational input can be verified automatically against specifications definable in a rich fragment of FBTL. On the other hand, we have shown that basic safety and liveness properties—like reachability of a safe state and persistency in safe states—become undecidable if functions (or some other data structures suitable for encoding bit-strings) occur in the input of sequential nullary ASMs, or if the computational power of sequential nullary ASMs with relational input is enhanced in a straightforward manner. Altogether the results in Chapter 1 might suggest that with sequential nullary ASMs we are approaching the limit of automatic verifiability of ASMs.

It is worth noticing that the simple programming language induced by sequential nullary ASMs goes beyond the scope of finite state systems. Equipped with the appropriate features, our simple programming language is more expressive than typical description languages for finite state systems (like, e.g., SMV [McM93]). In fact, one can argue that the automatic verifier implicitly described

in the proof of Theorem 1.3.3 performs symbolic model checking of software. (Recall in this context our brief discussion of software problems in the introduction to Chapter 1). As a possible field of application for sequential nullary ASMs we propose the high-level ASM descriptions that naturally occur when designing complex dynamic systems by means of the ASM method. The automatic verifier we have provided could support the verification of such high-level ASM descriptions.

ASM relational transducers, which we have been concerned with in Chapter 2, are interactive ASMs specially tailored for the specification of electronic commerce protocols. We have shown that several verification problems related to electronic commerce applications are decidable for (restricted) ASM relational transducers. Notice that this result does not contradict the negative results in Chapter 1 concerning the automatic verifiability of ASMs more powerful than sequential nullary ASMs, because our restricted ASM transducers can be regarded as parallel-composed sequential nullary ASMs.

A still open problem concerning ASM transducers is the verification of systems of interacting relational transducers [AVFY98]. We are optimistic that our investigation in Chapter 2 can serve as a basis for future research in this direction. Another open problem concerns efficiently verifiable relational transducer. Polynomial-space complexity is too expensive for large-scale applications, especially when it comes to verification with respect to a given database. The question is whether there are further restrictions of ASM transducers (with input-bounded quantification) such that the corresponding verification problems can be solved efficiently.

II

Choiceless Complexity

5

Logarithmic-Space Reducibility via Abstract State Machines

Our investigations in the first part were motivated by applications of ASMs to practical computer science. In this second part, we study *applications of ASMs to the theory of computation*. We start below by introducing an ASM model specially tailored for describing logarithmic-space computable functions from structures to structures. Our model can serve as a basis for a *reduction theory among structures* and furthermore leads to the definition of a new complexity class, called *Choiceless Logarithmic Space*. In the next chapter, we compare this new class with (standard) Logarithmic Space and Choiceless Polynomial Time. The latter complexity class has recently been defined by Blass, Gurevich, and Shelah [BGS99].

Logarithmic-space computability (or *logspace computability* for brevity) is an important level of complexity, for several reasons:

- It can be viewed as the natural notion of computability with 'very little' memory.
- Logspace-computable functions are computable in parallel very efficiently (i.e., in polylogarithmic time) with a reasonable amount of hardware (i.e., by circuits of polynomial size).
- Logspace reductions are widely accepted as a natural basis for completeness results for important complexity classes like PTIME, NPTIME, and PSPACE. Indeed, almost all complete problems for these classes are complete with respect to logspace reductions (see, e.g., [Pap94, GHR95]).

By definition, a function from strings to strings is logspace-computable if it is computable by a Turing machine which, on inputs of length n, uses at most $O(\log n)$ cells of its work tapes. Since many computational problems arising in computer science and logic have input instances that are naturally viewed as structures rather than strings, one usually lifts this definition to functions from structures to structures by encoding structures as stings and by focusing on Turing machines that manipulate sting representations of structures. In this way one immediately obtains a notion of *logspace computability on structures* (which subsumes the usual notion of logspace computability on strings) and, based on that notion, *reductions among structures* (that are reductions between computational problems whose input instances are structures).

There is, however, a subtle but important disadvantage to this lifting of reductions by means of encoding: it is not known whether there exists an 'easily' computable string representation of structures such that the obtained notion of logspace computability on structures satisfies the following two conditions:

- Any function f logspace-computable according to this notion is isomorphism-invariant (i.e., whenever $\mathcal{A}$ and $\mathcal{A}'$ are isomorphic structures in the domain of f, then $f(\mathcal{A})$ and $f(\mathcal{A}')$ are also isomorphic).
- The set of (representations of) functions logspace-computable according to this notion is recursive.

But in order to serve as a basis for a reduction theory among structures our notion of logspace computability on structures should satisfy at least these two conditions. (Further conditions are formulated below.) The situation calls for a computation model that deals with structures directly rather than via string encodings, and in particular, for a notion of logspace computability defined by means of such a model.

Aside from ASMs, several other computation models on structures have been proposed in the literature (see [AHV95, BGS99, BGVdB99, Gur99] and the references there), in particular the *generic machines* of Abiteboul and Vianu [AV91]. Both ASMs and generic machines are computationally complete: they can calculate all (isomorphism-invariant) functions on structures that are computable in the usual sense, i.e., via string encodings. Hence, the notion of computable function on structures is well-understood. The situation becomes much more intriguing on lower complexity levels. For instance, it is not known whether there exists a computation model on structures (e.g., a class of restricted ASMs) such that the machines of this model compute precisely the class of all logspace-computable functions. (This question is related to the question of existence of an easily computable string representation of structures mentioned above. It disappears on ordered structures.) A similar problem arises on the polynomial-time level: recall from the introduction that it is not known whether there exists a query language in which precisely all polynomial-time computable queries are

expressible. The straightforward solution, namely to impose suitable time or space restrictions on one of the known, computationally complete machine models, failed so far due to the lack of appropriate time and space measures. For all known (honest) time and (natural) space measures the computational completeness of these models does not scale down to lower complexity levels. In particular, the notion of reduction among structures that we want to put forward in this chapter and which is based on ASMs does not capture all logspace reductions among (unordered) structures.

An alternative notion of reduction among structures is provided by *first-order interpretations* [Hod93, Chapter 5]. Informally, a first-order interpretation of a structure $\mathcal{A}$ in a structure $\mathcal{B}$ is given by a collection of FO formulas that define an isomorphic copy of $\mathcal{A}$ inside $\mathcal{B}$. It is well-known that first-order interpretations are weaker than logspace reductions, even on ordered structures. One way to enhance the power of interpretations is to consider (FO+DTC) *interpretations* instead (that are interpretations defined by means of (FO+DTC) formulas). By a result due to Immerman, (FO+DTC) interpretations can express precisely all logspace reductions among ordered structures [Imm87]. Moreover, once a reduction among structures is defined in terms of (FO+DTC) it is guaranteed to be logspace-computable. There is no need to analyze the storage requirements of an algorithm. Unfortunately, handling (FO+DTC) is not as straightforward as, say, a programming language and requires a certain familiarity with logic. More to the point, expressions in (FO+DTC) tend to be rather complex and hard to read when describing non-trivial reductions. Finally, if no linear order is available, then (FO+DTC) interpretations also fail to express all logspace reductions among structures.

After these introductory remarks, we can now formulate some necessary conditions for the notion of reduction among structures that we want to put forward:

1. Reductions should be computable in logarithmic space.

2. Reasonable closure properties should be satisfied. In particular, the class of reductions should be closed under composition.

3. Reductions should have at least the power of (FO+DTC) interpretations.

Our notion of reduction among structures is based on *bounded-memory ASMs*, that are ASMs specially tailored for describing logspace-computable functions from structures to structures. To establish a reduction among structures it suffices to present a bounded-memory ASM that computes the reduction. As in the case of (FO+DTC) interpretations there is no need to analyze the storage requirements of the presented ASM for it is guaranteed to run in logarithmic space. Furthermore, any reduction defined by means of bounded-memory ASMs is isomorphism-invariant by definition of ASMs, and the set of (representations of) bounded-memory ASMs is recursive.

The reader may wonder why we call our ASMs "bounded-memory ASMs" rather than, say, "logspace ASMs". Recall that ASMs have been put forward as a model for describing algorithms on their natural level of abstraction [Gur95]. When we describe common logspace algorithms on their natural abstraction level, we observe that most of them actually use a *bounded* number of *memory* locations (e.g., variables), each of which stores an object that is identified by a logarithmic number of bits (e.g., an element of the input structure or a natural number polynomially bounded in the cardinality of the input structure). Since the term "logspace" refers to the bit-level, we find the name "bounded-memory ASMs" more adequate for the spirit of ASMs.

Actually, we introduce two ASM models for describing logspace-computable functions on structures. ASMs of the first model, which we call *nullary ASMs*, are essentially basic ASMs in the sense of [Gur97b], all of whose dynamic symbols are nullary. Although nullary ASMs already suffice to describe all logspace-computable functions on ordered structures, they are fairly weak without access to a linear order. The more powerful bounded-memory ASMs are obtained from nullary ASMs by adding sequential composition and distributed execution (see Section 5.3). They have two main advantages over nullary ASMs. Firstly, even for reductions that can be defined by means of nullary ASMs, bounded-memory ASMs often admit presentations that are more succinct and easier to understand. Secondly, bounded-memory ASMs are strictly more expressive than both nullary ASMs and the logic (FO+DTC).

Bounded-memory ASMs can be viewed as instances of an ASM model that has recently been introduced by Blass, Gurevich, and Shelah while investigating the *choiceless fragment of* PTIME [BGS99]. Let us briefly recall their motivation for introducing their model, thereby explaining an important feature of our model. An important feature of algorithms on ordered structures (e.g., strings) is that they can make choices: out of any set of elements of the input structure they can select one element, say, the smallest one, and proceed from there. Typical examples are graph algorithms. Consider, for instance, the following common reachability algorithm which, on input of a finite ordered graph $\mathcal{G}$ and a distinguished node *source* in $\mathcal{G}$, computes the set of all nodes reachable from *source*. The algorithm maintains an auxiliary set X of frontier nodes. Initially, $X = \{source\}$. In every step, the algorithm chooses a node in X, say, by selecting the smallest one, adds the neighbors of this node to the set of reachable nodes, and updates the frontier set X accordingly. The algorithm terminates when the set of reachable nodes becomes stable. It is easy to see that in this example explicit choice is not really needed. Instead of choosing one particular frontier node, the algorithm could also process all of them in parallel. (It seems, however, that this is not always possible. For example, in matching algorithms the ability to make choices is used in a much more sophisticated way. In fact, there is no known efficient algorithm without explicit choice that solves the perfect matching problem for bipartite graphs.)

The main idea of Blass, Gurevich, and Shelah in [BGS99] is to replace explicit choice with parallel computations over hereditarily finite sets. To this end, they introduced an ASM model that can form and handle hereditarily finite sets. Inspired by their model, we equip bounded-memory ASMs with the ability to form and handle 'small' sets. This enables our ASMs to invent new elements by simply forming sets of already known elements or sets. The newly formed sets can then serve as new elements. (Observe that, while reducing an instance of one computational problem to an instance of another such problem, one often has to invent new elements, e.g., new nodes in a graph.) The introduction of hereditarily finite sets has further consequences: our ASMs are, like those of Blass, Gurevich, and Shelah, choiceless in the sense that they can form a set without ever actually choosing one *particular* element of the set. Indeed, the class of problems computable by means of bounded-memory ASMs can be seen as the *choiceless fragment of* Logspace. We compare this fragment with the choiceless fragment of Ptime in the next chapter.

Related Work. Our ASM model is based on the (computationally complete version of the) ASM model developed by Blass, Gurevich, and Shelah in [BGS99]. There the reader will also find references to other computation models using hereditarily finite sets as a natural domain for computations. Note that the subject of object invention has also been investigated in the context of query languages (see [AHV95, BGVdB99] and the references there).

Outline of the Chapter. In Section 5.1, we provide the set-theoretic and logical framework for our ASM model. In Sections 5.2 and 5.3, we introduce nullary and bounded-memory ASMs, respectively, investigate their computational power, and present an example of a reduction via bounded-memory ASMs. In Section 5.4, we show that termination of every bounded-memory ASM can be guaranteed by a syntactic manipulation of its program. This motivates the definition of the already mentioned choiceless fragment of Logspace.

5.1 Hereditarily Finite Sets as a Domain for Computation

We recall some basic definitions from [BGS99] and introduce the notion of *HF-initialized ASM*. Each state of an HF-initialized ASM contains all hereditarily finite sets built from the elements of the current input domain. The main purpose of these sets is to serve as auxiliary storage during a computation. We also introduce a logic for expressing properties of states of HF-initialized ASMs. The formulas of this logic will serve as guards in our ASM model.

HF-initialized ASMs

For the reader's convenience we first recall the notion of hereditarily finite set. Let A be a finite set of atoms (that are elements which are not sets, also called urelements in set theory). The set $\mathrm{HF}(A)$ of *hereditarily finite sets over* A is the least set H such that every finite subset of $A \cup H$ is an element of H.

Consider a set $X \in \mathrm{HF}(A)$. X is called *transitive* if, whenever $Z \in Y \in X$, then $Z \in X$. $\mathrm{TC}(X)$ denotes the *transitive closure* of X, i.e., the least transitive set Y with $X \in Y$. Obviously, $\mathrm{TC}(X)$ is finite and an element of $\mathrm{HF}(A)$. By the *size* of X we mean the cardinality of $\mathrm{TC}(X)$. The size of X is an upper bound for both the cardinality of $A \cap \mathrm{TC}(X)$ and the length k of chains $Y_1 \in Y_2 \in \ldots \in Y_k \in X$. The maximum k is also called the *rank* of X. For every natural number s, $\mathrm{HF}_s(A)$ denotes the restriction of $\mathrm{HF}(A)$ to sets of size at most s.

Proviso. In the remainder of the thesis, the universe of any finite structure is assumed to be a set of atoms. It is furthermore assumed that relational vocabularies do not contain the binary relation symbol $\in$ nor any constant symbol. □

Definition 5.1.1. Let Υ be a relational vocabulary and let $\mathcal{A}$ be a finite structure over Υ with universe A. Set $\Upsilon^+ = \Upsilon \cup \{\in, \mathit{unique}, \bigcup, \varnothing, \mathit{atoms}\}$ where $\in$ is a binary relation symbol, *unique* and $\bigcup$ are unary function symbols, and *atoms* and $\varnothing$ are constant symbols. The *HF-extension* of $\mathcal{A}$, denoted $\mathcal{A}^+$, is a structure over Υ^+ defined as follows:

- The universe of $\mathcal{A}^+$ is $A \cup \mathrm{HF}(A)$.
- Every $R \in \Upsilon$ is interpreted as in $\mathcal{A}$.
- $\in$ and $\varnothing$ are interpreted in the obvious way. *atoms* is interpreted as the set A. If X is a singleton set, then $\mathit{unique}(X)$ is the unique element of X; otherwise $\mathit{unique}(X) = \varnothing$. If a is an atom, i.e., an element of $\mathcal{A}$, then $\bigcup a = \varnothing$. If X is a set, say, $X = \{a_1, \ldots, a_k, Y_1, \ldots, Y_l\}$ where $a_1, \ldots, a_k$ are atoms and $Y_1, \ldots, Y_l$ are sets, then $\bigcup X = Y_1 \cup \ldots \cup Y_l$.

□

We consider ASMs whose states are HF-extensions of input structures (enriched with dynamic relations and functions).

Definition 5.1.2. Let $M = (\Upsilon, \mathit{initial}, \Pi)$ be an ASM where Υ_{in} and Υ_{out} are relational vocabularies and Υ_{stat} is $\varnothing^+$ (i.e., $\{\in, \mathit{unique}, \bigcup, \varnothing, \mathit{atoms}\}$). M is *HF-initialized* if for every input $\mathcal{I}$ appropriate for M, $\mathit{initial}(\mathcal{I})|_{\mathrm{in,stat}} = \mathcal{I}^+$ and in $\mathit{initial}(\mathcal{I})|_{\mathrm{dyn,out}}$, all relations are empty and all functions have range $\{\varnothing\}$. □

Observe that HF-initialized ASMs are uniquely determined by their vocabulary and program. This justifies to denote an HF-initialized ASM $(\Upsilon, \mathit{initial}, \Pi)$ by the pair (Υ, Π). We will do so from now on.

Remark 5.1.3. (a) In [BGS99] the authors considered HF-initialized ASMs that are equipped with an additional static function which maps any two objects to the set consisting of these two objects. For technical reasons, we make this function part of a term calculus defined below (see Definition 5.1.4).

(b) The function $\bigcup$ will only be important in the next chapter where we compare the ASM model of [BGS99] with our model. The reader may ignored this function throughout this chapter. □

The Logic (FO+BS)

First-order logic is not a suitable logic for the guards of HF-initialized ASMs because it allows us to express non-recursive operations. (Notice that every HF-extension of a finite structure contains the natural numbers in the form of the von Neumann ordinals $\varnothing$, $\{\varnothing\}$, $\{\varnothing, \{\varnothing\}\}, \dots$) Inspired by a term calculus introduced in [BGS99], we define a logic whose formulas will serve as guards in our ASM model. Although the logic is not as expressive as first-order logic on HF-extensions, it is strictly more expressive than first-order logic on input structures for it can handle 'bounded' sets. To emphasize this we denote the logic (FO+BS) where 'BS' alludes to 'bounded sets'.

Definition 5.1.4. *Terms* and *formulas* of the logic (FO+BS) are defined by simultaneous induction:

(T1) Every variable is a term.

(T2) The constant symbol $\varnothing$ is a term, and if t is a term, then $\mathit{unique}(t)$ is also a term.

(T3) If $t_1, \dots, t_k$ are terms and s is a natural number, then $\{t_1, \dots, t_k\}_s$ is a term.

(T4) If t is a term, x is a variable, φ is a formula, s is a natural number, and r is either *atoms* or a term without free occurrences of x, then $\{t : x \in r : \varphi\}_s$ is a term.

(F1) Every atomic formula (built from terms as usual) is a formula.

(F2) If φ and ψ are formulas, then $\varphi \vee \psi$ and $\neg\varphi$ are formulas.

The *free* and *bound variables* of (FO+BS) terms and formulas are defined in the obvious way. In particular, a variable occurs free in $\{t : x \in r : \varphi\}_s$ if it is different from x and occurs free in t, r or φ; x itself occurs bound.

Terms of the form $\{t_1, \dots, t_k\}_s$ or $\{t : x \in r : \varphi\}_s$ are also referred to as *set terms*. The *maximal size bound* of a (FO+BS) term t (resp. formula φ) is the maximum of 1 and all subscripts at sub-set-terms of t (resp. φ). □

To easy notation we frequently omit the subscripts at set terms when the description of the sets implies an obvious bound on their size.

Semantics of (FO+BS) Formulas. We define the semantics of (FO+BS) terms and formulas only with respect to HF-extensions of finite structures. Let Υ be a relational vocabulary and let $\mathcal{A}^+$ be the HF-extension of a finite structure $\mathcal{A}$ over Υ. For the sake of a uniform treatment, we view formulas as terms, i.e., we regard $\neg$, $\vee$, $=$, $\in$, and all relation symbols in Υ as function symbols denoting functions with range $\{\mathit{true}, \mathit{false}\}$. Consider a (FO+BS) term t over Υ^+ with $\mathrm{free}(t) \subseteq \{\bar{x}\}$. The interpretation of t in $\mathcal{A}^+$, denoted $t^{\mathcal{A}^+}$, is defined by induction on the construction of t, simultaneously for all interpretations $\bar{a}$ of $\bar{x}$ (chosen from the universe of $\mathcal{A}^+$):

- If t is a variable or a constant symbol, or if t is obtained by means of application of a function symbol, then $t^{\mathcal{A}^+}$ is defined as usual.
- Suppose that $t = \{t_1, \ldots, t_k\}_s$. If the set $\{t_1^{\mathcal{A}^+}[\bar{a}], \ldots, t_k^{\mathcal{A}^+}[\bar{a}]\}$ has size $\leq s$, then $t^{\mathcal{A}^+}[\bar{a}]$ is this set; otherwise, $t^{\mathcal{A}^+}[\bar{a}] := \varnothing$.
- Suppose that $t = \{t_0 : y \in r : \varphi\}_s$ where $\mathrm{free}(t_0) \cup \mathrm{free}(\varphi) \subseteq \{\bar{x}, y\}$. If the set $\{t_0^{\mathcal{A}^+}[\bar{a}, b] : b \in r^{\mathcal{A}^+}[\bar{a}] : \mathcal{A}^+ \models \varphi[\bar{a}, b]\}$ has size $\leq s$, then $t^{\mathcal{A}^+}[\bar{a}]$ is this set; otherwise, $t^{\mathcal{A}^+}[\bar{a}] := \varnothing$.

This concludes the definition of the semantics of (FO+BS) terms and formulas.

Although the definition of (FO+BS) does not mention quantification, there is a bounded version of quantification definable in (FO+BS). For example, let $(\exists x \in r)\varphi$ abbreviate the (FO+BS) formula $(\varnothing \in \{\varnothing : x \in r : \varphi\})$. (Technically, the set term in this formula needs a subscript to bound its size; 2 will do.) By using *atoms* for the range term r in $(\exists x \in r)\varphi$ we can simulate quantification over all atoms (or, in the case of HF-initialized ASMs, over all elements of an input structure). This is the main idea in the proof of the following lemma.

Lemma 5.1.5. *Let Υ be a relational vocabulary. For every* FO *sentence φ over Υ there exists a* (FO+BS) *sentence φ^+ over Υ^+ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, $\mathcal{A} \models \varphi$ iff $\mathcal{A}^+ \models \varphi^+$.*

The next lemma shows that every (FO+BS) formula can be evaluated in Logspace.

Lemma 5.1.6. *Let Υ be a relational vocabulary. For every* (FO+BS) *sentence φ over Υ^+ there exists a logspace-bounded Turing machine which, for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, on input (of an encoding of) $\mathcal{A}$, decides whether $\mathcal{A}^+ \models \varphi$.*

Proof. Let s be the maximal size bound of φ. An easy induction on the construction of φ shows that all sets that we need to consider in order to decide $\mathcal{A}^+ \models \varphi$ have size $\leq s$.

Claim. Every set $X \in \mathrm{HF}_s(A)$ can be stored in space $O(\log |A|)$.

Proof of the claim. Every $X \in \mathrm{HF}_s(A)$ can be represented as a tree with node set $\mathrm{TC}(X)$ and root X: let there be an edge from Y to Z iff $Y \in Z$. A leaf is either an atom or the empty set. This tree has $|\mathrm{TC}(X)| \leq s$ nodes and can be stored in space $s \log |A| + s^2$. To see this, observe that X contains at most s atoms, each of which needs space $\log |A|$. The structure of the tree can be stored as a binary relation over $\{1, \ldots, s\}$ in space s^2. $\Diamond$

As in the definition of the semantics of (FO+BS) formulas, we view φ as a term. It suffices to show that for every (FO+BS) term t over Υ^+ with $\mathrm{free}(t) = \{\bar{x}\}$ there exists a logspace-bounded Turing machine $M_{\Upsilon,t}$ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ and all interpretations $\bar{a}$ of $\bar{x}$ (chosen from the universe of $\mathcal{A}^+$), $M_{\Upsilon,t}$ on input of an encoding of $(\mathcal{A}, \bar{a})$, outputs an encoding of $t^{\mathcal{A}^+}[\bar{a}]$. The construction of $M_{\Upsilon,t}$ is by an obvious induction on the construction of t. We omit the details. □

5.2 A Restricted Model

In this section we introduce *nullary ASMs*, a restricted model for LOGSPACE computability on structures. Despite their simplicity, nullary ASMs already suffice to describe all LOGSPACE-computable functions on ordered structures.

Definition 5.2.1. Fix an ASM vocabulary Υ where Υ_{in} and Υ_{out} are relational vocabularies, $\Upsilon_{\mathrm{stat}} = \varnothing^+$, and Υ_{dyn} contains only nullary function symbols and the distinguished set symbol *Universe*. Set $\Upsilon^- = \Upsilon^+ - (\Upsilon_{\mathrm{out}} \cup \{\mathit{Universe}\})$. *Nullary programs* are defined inductively:

- **Updates:** For every relation symbol $R \in \Upsilon_{\mathrm{dyn}} \cup \Upsilon_{\mathrm{out}}$, every nullary function symbol $v \in \Upsilon_{\mathrm{dyn}}$, and all (FO+BS) terms $\bar{t}, t_0$ over Υ^-, each of the following is a nullary program: $R(\bar{t})$, $v := t_0$.

- **Conditionals:** If Π is a nullary program and φ is a (FO+BS) formula over Υ^-, then (`if` φ `then` Π) is a nullary program.

- **Parallel composition:** If Π_0 and Π_1 are nullary programs, then $(\Pi_0 || \Pi_1)$ is a nullary program.

The *free* and *bound variables* of a nullary program are defined in the obvious way. The *maximal size bound* of a nullary program is the maximum of 1 and all maximal size bounds of (FO+BS) terms and formulas occurring in the program.

A *nullary ASM* is an HF-initialized ASM whose program is a nullary program (without free variables). □

Notice that, similar to logspace-bounded Turing machines, the output of a nullary ASM is write-only: once a tuple is put into an output relation it remains there for the rest of the computation. Because of this, the output of nullary ASMs is in general relational. Since we will later also consider sequential compositions of nullary ASMs, it is natural to focus our attention on nullary ASMs with relational input and output vocabularies. Notice also that this restriction has no practical impact. For example, instead of a function f one may include the graph G_f of f in an input structure. A nullary ASM can then access the value of f at node a by means of the (FO+BS) term $\mathit{unique}\{y \in \mathit{atoms} : G_f(a, y)\}$.

Remark 5.2.2. (a) We excluded boolean symbols from the dynamic vocabulary of nullary ASMs to simplify the presentation of technicalities in later sections. Nevertheless, one can easily mimic a boolean variable by restricting the range of a nullary dynamic function to $\{\mathit{true}, \mathit{false}\}$, where *false* stands for the empty set and *true* for the set $\{\varnothing\}$.

(b) Nullary ASMs are not to be confused with the sequential nullary ASMs considered in Chapter 1. In particular, nullary ASMs are neither sequential nor non-deterministic. The former is due to the expressive power of (FO+BS). For instance, a guard of the form $(\forall x \in \mathit{atoms})\varphi(x)$ expresses a massively parallel search of the input universe. □

Outputs of Nullary ASMs. Consider a nullary ASM $M = (\Upsilon, \Pi)$. Let $\mathcal{I}$ be an input appropriate for M and suppose that $\rho = (\mathcal{S}_i)_{i\in\omega}$ is the run of M on $\mathcal{I}$. We say that M *halts* on $\mathcal{I}$ if $\mathcal{S}_j = \mathcal{S}_{j+1}$ for some $j \in \omega$. Notice that in this case, $\mathcal{S}_j = \mathcal{S}_i$ for every $i \in \omega$ satisfying $\mathcal{S}_i = \mathcal{S}_{i+1}$ because M is deterministic. Suppose that M halts on $\mathcal{I}$. Choose $j \in \omega$ such that $\mathcal{S}_j = \mathcal{S}_{j+1}$ and let $\mathcal{S}_j^U$ denote the substructure of $\mathcal{S}_j$ induced by $\mathit{Universe}^{\mathcal{S}_j}$. Define the *output* of M on $\mathcal{I}$ to be the structure $\mathcal{S}_j^U|_{\mathrm{out}}$, i.e., the reduct of $\mathcal{S}_j^U$ to the vocabulary Υ_{out}. This convention enables us to define the universe of the output structure by putting all those elements into *Universe* which we want to be present in the output.

Notice that the output structure of a nullary ASM is always finite. To see this observe that, if s is the maximal size bound of the program of a nullary ASM M, then s is an upper bound for the size of the sets that M can examine. In particular, if M halts on an input $\mathcal{I}$ with output $\mathcal{O}$, then the universe of $\mathcal{O}$ is a subset of $I \cup \mathrm{HF}_s(I)$. For a finite I, $\mathrm{HF}_s(I)$ is obviously finite. Observe also that, in general, $\mathcal{O}$ cannot serve as input structure to another nullary ASM because the universe of $\mathcal{O}$ may contain elements that are sets. To avoid problems when we compose nullary ASMs later on, we will from now on assume that the output structures of nullary ASMs have their non-atomic elements converted to genuine atoms.

Example 5.2.3. For a binary relation E we denote by $DTC(E)$ the *deterministic transitive closure* of E, i.e., the transitive reflexive closure of the deterministic

part of E. (The deterministic part of E is obtained from E by removing all edges that start at a node with out-degree ≥ 2.) Now consider the binary query Q_{DTC} which maps every finite ordered directed graph $\mathcal{G} = (V, E, <)$ to $DTC(E)$. We are going to define a nullary ASM that computes Q_{DTC} in the sense that, on input $\mathcal{G}$, it outputs the graph $(V, DTC(E))$.

Let us first concentrate on the following instance of the problem. Write a nullary program Π which, given a node *startNode* $\in V$, outputs all nodes on the deterministic E-path that starts at *startNode*. Here is a possible solution. In the first step, Π initializes a nullary dynamic function *pebble* with *startNode*. Then, in all subsequent steps, Π outputs *pebble* and moves *pebble* along the deterministic path by executing the update $pebble := succ'(pebble)$, where

$$succ'(x) \quad := \quad unique\{y \in atoms : E(x, y)\}.$$

But how do we ensure termination of this process if the path leads into a cycle? Every cycle in $\mathcal{G}$ has at most $|V|$ nodes. Hence, it suffices to set up a counter that triggers termination after $|V|$ steps. Let *counter* be a nullary dynamic function and let *least* be the least node in $\mathcal{G}$ with respect to $<$. (*least* can be defined by means of the term $unique\{x \in atoms : \neg(\exists y \in atoms)\, y < x\}$.) Π initializes *counter* with *least* and executes in every step the update $counter := counter + 1$, where

$$\begin{aligned} counter + 1 \quad := \quad & unique\{x \in atoms : \\ & \qquad x > counter \wedge \\ & \qquad (\forall y \in atoms)(y > counter \rightarrow y \geq x)\}. \end{aligned}$$

Π is defined below. For clarity, we write *initializePebble*, *movePebble*, and *nextPath* instead of the terms $\varnothing$, $\{\varnothing\}$, and $\{\{\varnothing\}\}$, respectively. Intuitively, Π outputs a pair $(startNode, a)$ for every node a on the deterministic path that starts at *startNode*. It becomes idle if *pebble* has no unique E-successor or if *counter* assumes the 'value' $\varnothing$ after $|V|$ steps.

`program` Π:

```
if mode = initializePebble then
    pebble := startNode
    counter := least
    mode := movePebble

if mode = movePebble then
    if pebble ≠ ∅ ∧ counter ≠ ∅ then
        DTC(startNode, pebble)
        pebble := succ'(pebble)
        counter := counter + 1
    else
        mode := nextPath
```

From Π one can now easily obtain a nullary ASM $M_{\mathrm{DTC}} = (\Upsilon, \Pi_{\mathrm{DTC}})$ that computes Q_{DTC}. Let the ASM vocabulary Υ be defined by $\Upsilon_{\mathrm{in}} := \{E, <\}$, $\Upsilon_{\mathrm{stat}} := \varnothing^+$, $\Upsilon_{\mathrm{dyn}} := \{\mathit{mode}, \mathit{pebble}, \mathit{counter}, \mathit{startNode}, \mathit{Universe}\}$, and $\Upsilon_{\mathrm{out}} := \{DTC\}$. Π_{DTC} systematically varies *startNode* over all nodes in V and calls for each instance of *startNode* the above program Π (see line (1) below). When Π terminates, Π_{DTC} resumes its computation in line (2).

`program` Π_{DTC}:

`if` $\mathit{mode} = \mathit{initial}$ `then`
  $\mathit{startNode} := \mathit{least}$
  $\mathit{mode} := \mathit{initializePebble}$

Π (1)

`if` $\mathit{mode} = \mathit{nextPath} \wedge \mathit{startNode} \neq \varnothing$ `then` (2)
  $\mathit{Universe}(\mathit{startNode})$
  $\mathit{startNode} := \mathit{startNode} + 1$
  $\mathit{mode} := \mathit{initializePebble}$

□

It is worth noticing that there is no obvious way to define a nullary ASM that computes $DTC(E)$ without access to a linear order on the nodes of the input graph. Without such an order it is not obvious how to count—and in this way detect a cycle—nor how to maneuver *pebble* from one node to another so that all nodes of $\mathcal{G}$ are pebbled at least once. The example reveals two defects of nullary ASMs:

1. It is not clear how to ensure **termination** of nullary ASMs on unordered input structures. (As an aside, note that one could easily solve this problem by modifying the definition of the output of a nullary ASM. Every run of a nullary ASM M consists of only finitely many different states. Since the output of M is write-only, the output become stationary after finitely many computation steps. Hence, one could terminate a computation of M once the output is stationary. However, we do not consider this a satisfactory solution.)

2. If no linear order is available, then nullary ASMs may not be able to **systematically explore** their input. For example, consider the unary query Q_S defined by the FO formula $S(x)$. On input $\mathcal{I} = (I, S^{\mathcal{I}})$, a nullary ASM computing Q_S would only have to copy the set $S^{\mathcal{I}}$ to some output set. However, one can show that no such nullary ASM exists. (If $S^{\mathcal{I}}$ and $I - S^{\mathcal{I}}$ are large enough, then the range of any nullary dynamic function is $\mathrm{HF}(\varnothing)$.) This shows that there are simple FO definable queries on unordered structures which cannot be computed by any nullary ASM.

In the next section, we are going to cure both defects by upgrading nullary ASMs to bounded-memory ASMs (see Theorem 5.4.1 and Lemma 5.3.5).

Computational Power of Nullary ASMs.

We compare the computational power of nullary ASMs with the computational power of deterministic Turing machines. Consider a nullary ASM M of vocabulary Υ. *M computes* a partial function from $\mathrm{Fin}(\Upsilon_{\mathrm{in}})$ to $\mathrm{Fin}(\Upsilon_{\mathrm{out}})$ in the obvious way. If $\Upsilon_{\mathrm{out}} = \{\mathit{accept}\}$ and $C \subseteq \mathrm{Fin}(\Upsilon_{\mathrm{in}})$ is the class of structures on which M halts, then we regard the partial function computed by M as a boolean query on C.

Lemma 5.2.4. *(1) Every function computable by a nullary ASM is* Logspace*-computable. (2) On ordered input structures, nullary ASMs compute precisely the class of* Logspace*-computable boolean queries.*

Proof. To (1). We show that any nullary ASM $M = (\Upsilon, \Pi)$ can be simulated by a logspace-bounded Turing machine. Let s be the maximal size bound of Π and let $v_1, \dots, v_d$ be an enumeration of the nullary dynamic function symbols in Υ_{dyn}. Consider a run $\rho = (\mathcal{S}_i)_{i\in\omega}$ of M. For every $i \in \omega$, $\mathcal{S}_{i+1}$ is entirely determined by the values $v_1^{\mathcal{S}_i}, \dots, v_d^{\mathcal{S}_i}$ and the input component $\mathcal{S}_0|_{\mathrm{in}}$. (Recall that output symbols do not occur in guards and thus do not influence a computation of M.) Furthermore, if I is the universe of the input component $\mathcal{S}_0|_{\mathrm{in}}$, then $v_1^{\mathcal{S}_i}, \dots, v_d^{\mathcal{S}_i} \in I \cup \mathrm{HF}_s(I)$. By the claim in the proof of Lemma 5.1.6 we know that every $X \in \mathrm{HF}_s(I)$ can be stored in space $O(\log |I|)$. It is now a matter of patience to construct a Turing machine which for every $\mathcal{I} \in \mathrm{Fin}(\Upsilon_{\mathrm{in}})$, on input (of an encoding of) $\mathcal{I}$, simulates M on $\mathcal{I}$ using its work tape to store (encodings of) interpretations of $v_1, \dots, v_d$ (and maybe for bookkeeping some additional counters whose length is logarithmically bounded in the size of $\mathcal{I}$).

To (2). Let Q be boolean query on $\mathrm{Fin}(\Upsilon)$. If Q is computable by a nullary ASM, then it is Logspace-computable by assertion (1). Now suppose that Q is Logspace-computable. By a well-known result due to Immerman [Imm87], we can assume that Q is definable in (FO+DTC). Hence, it suffices to display for every formula $\varphi \in (\mathrm{FO+DTC})(\Upsilon)$ with $\mathrm{free}(\varphi) = \{\bar{x}\}$ a nullary ASM M_φ with input vocabulary $\Upsilon \cup \{\bar{x}\}$, such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ and all interpretations $\bar{a}$ of $\bar{x}$, M_φ accepts $(\mathcal{A}, \bar{a})$ iff $\mathcal{A} \models \varphi[\bar{a}]$. The construction of M_φ is by induction on the construction of φ. We already did most of the work in Example 5.2.3; the details are left to the reader. □

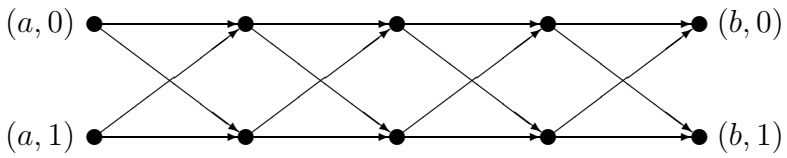
The next lemma contrasts our previous observation that there are simple FO definable queries on unordered structures which are not computable by nullary ASMs.

Lemma 5.2.5. *There exist a class C of (unordered) graphs and a nullary ASM M so that M computes a boolean query on C which is not definable in deterministic transitive-closure logic* (FO+DTC).

Proof. For every finite directed graph $\mathcal{G} = (V, E)$, define the doubled version of $\mathcal{G}$, $2\mathcal{G} = (2V, 2E)$, by setting $2V = V \times \{0, 1\}$ and

$$2E \;=\; \big\{\big((a,i),(b,j)\big) : (a,b) \in E \text{ and } i,j \in \{0,1\}\big\}.$$

For instance, if $\mathcal{G}$ is a path of length 4 from a node a to a note b, then $2\mathcal{G}$ is the following graph:



The class C is the union of two classes C_1 and C_2 of double graphs. To the definition of C_1. Consider a graph $\mathcal{G}$ consisting of two disjoint (directed) cycles of the same even diameter. Obtain $2\mathcal{G}'$ from $2\mathcal{G}$ by labeling two nodes in $2\mathcal{G}$ as follows:

1. Choose an arbitrary node in $2\mathcal{G}$ and label it *source*.
2. Choose another node in the same connectivity component as *source* such that the distance between this node and *source* is maximal; label this node *target*.

Let C_1 be the collection of all $2\mathcal{G}'$. To the definition of C_2. Modify $2\mathcal{G}'$ to $2\mathcal{G}''$ by moving the label *target* to an arbitrary node in the other connectivity component of $2\mathcal{G}'$. Let C_2 be the collection of all $2\mathcal{G}''$. Obviously, there is a path from *source* to *target* in every graph in C_1. No graph in C_2 has this property. Let $C := C_1 \cup C_2$. Due to an observation by Immerman (see also [GM95]) there exists no (FO+DTC) sentence φ such that for all $\mathcal{G}^* \in C$, $\mathcal{G}^* \models \varphi$ iff $\mathcal{G}^* \in C_1$.

It remains to display a nullary ASM M that halts on every $\mathcal{G}^* \in C$ and accepts $\mathcal{G}^*$ iff $\mathcal{G}^* \in C_1$. The idea is to move two identical pebbles simultaneously around one doubled cycle, one on the 0-copy and one on the 1-copy of the cycle. Initially, M places both pebbles on the two successor nodes of *source*. Then, in every step, it moves both pebbles simultaneously to the successor nodes of the currently pebbled nodes. This continues until either *target* or *source* is pebbled. If *target* is pebbled, M accepts the input and halts. If *source* is pebbled, both pebbles have been moved around the doubled cycle without reaching *target*. In that case, M halts without accepting. To the definition of $M = (\Upsilon, \Pi)$. Let Υ be given by $\Upsilon_{\text{in}} := \{E, \mathit{source}, \mathit{target}\}$, $\Upsilon_{\text{stat}} := \varnothing^+$, $\Upsilon_{\text{dyn}} := \{\mathit{mode}, \mathit{pebbles}, \mathit{Universe}\}$, and $\Upsilon_{\text{out}} := \{\mathit{accept}\}$.

```
program Π:

  if mode = initializePebble then
     pebbles := {x ∈ atoms : E(source, x)}
     mode := movePebble

  if mode = movePebble then
     if target ∉ pebbles ∧ source ∉ pebbles then
        pebbles := {x ∈ atoms : (∃y ∈ pebbles)E(y, x)}
     else
        if target ∈ pebbles then accept
```

□

As pointed out in the discussion following Example 5.2.3, there is no obvious way to tell whether a given nullary ASM M halts on all input structures. However, on ordered input structures one can set up a counter (like *counter* in Example 5.2.3) that terminates M once the maximal number of possible configurations of M has been reached. This is the crux in the proof of the following lemma.

Lemma 5.2.6. *Every nullary ASM M whose input vocabulary contains $<$ can be altered by a syntactic manipulation so that the resulting nullary ASM M' has the same input and output vocabulary as M, halts on every (ordered) input, and produces the same output as M on every (ordered) input on which M halts.*

In the next, section we will upgrade nullary ASMs to bounded-memory ASMs. This will improve the handling of our ASM model in practice as well as its computational power (when no order is present).

5.3 Bounded-Memory ASMs

Nullary ASMs do not properly reflect two important properties of logspace-computable functions, namely that such functions are closed under *composition* and *distributed execution*. Nullary ASMs cannot simply be composed because output relations must not occur in guards. To see what we mean by distributed execution, consider a logspace-bounded Turing machine M that obtains (an encoding of) a finite graph $\mathcal{G}$ as input together with some parameter a, which is node in $\mathcal{G}$. In order to compute the output of M on $(\mathcal{G}, a)$ for every node a, we may execute all instances of $M(\mathcal{G}, x)$ in parallel on distributed processors. Obviously, there is no interference between computations for different nodes. This distributed execution is still in (sequential) LOGSPACE for we obtain the same result with a logspace-bounded Turing machine $N(\mathcal{G})$ that enumerates all node in $\mathcal{G}$ in some order and simulates $M(\mathcal{G}, a)$ for every node a in that order. We add both composition and distributed execution to nullary ASMs.

Definition 5.3.1. Fix an ASM vocabulary Υ where Υ_{in} and Υ_{out} are relational vocabularies and $\Upsilon_{\text{stat}} = \varnothing^+$. *Distributed programs* are obtained from nullary programs as follows:

- Every nullary program without free variables is a distributed program.
- **Distributed composition:** Let $x_1, \ldots, x_k$ be pairwise distinct variables and let Π be a nullary program all of whose free variables but none of whose bounded variables occur among $x_1, \ldots, x_k$. For each x_i, let r_i denote *atoms* or a closed (FO+BS) term over Υ^+_{in}. Obtain $\Pi_{\bar{x}}$ from Π by replacing every occurrence of a nullary dynamic function symbol v in Π with the term $v(x_1, \ldots, x_k)$. Then

  `do-for-all` $x_1 \in r_1, \ldots, x_k \in r_k$
  $\quad \Pi_{\bar{x}}$

  is a distributed program. (Notice that in this program all dynamic function symbol are now k-ary symbols.)
- **Guarded distributed composition:** Let $x_1, \ldots, x_k$, $r_1, \ldots, r_k$, Π, and $\Pi_{\bar{x}}$ be as in the above program-formation rule. In addition, let $\alpha_1, \ldots, \alpha_n$ be atomic FO formulas over Υ_{out} with $\text{free}(\alpha_i) \subseteq \{x_1, \ldots, x_k\}$ such that, if α_i has the form $R(\bar{t})$ and there is an update $R(\bar{t}')$ in Π, then $\bar{t} = \bar{t}'$ (see also Remark 5.3.7). Then

  `do-for-all` $x_1 \in r_1, \ldots, x_k \in r_k$
  `unless` $\alpha_1 \vee \ldots \vee \alpha_n$
  $\quad \Pi_{\bar{x}}$

  is a distributed program.

A *bounded-memory program* Π (over Υ) is a finite sequence $(\Pi_1, \ldots, \Pi_q)$ of distributed programs, each over some ASM vocabulary Υ^i, such that

- $\Upsilon_{\text{in}} = \Upsilon^1_{\text{in}}$,
- $\Upsilon_{\text{out}} \subseteq \Upsilon^1_{\text{out}} \cup \ldots \cup \Upsilon^q_{\text{out}}$,
- $\Upsilon^i_{\text{in}} = \Upsilon_{\text{in}} \cup \Upsilon^1_{\text{out}} \cup \ldots \cup \Upsilon^{i-1}_{\text{out}}$,
- $\Upsilon^i_{\text{dyn}} \cap \Upsilon^j_{\text{dyn}} = \{\mathit{Universe}\}$ if $i \neq j$, and
- $\Upsilon^i_{\text{dyn}} \cup \Upsilon^i_{\text{out}} \subseteq \Upsilon_{\text{dyn}} \cup \Upsilon_{\text{out}}$.

Each Π_i is called a *stratum* of Π.

A *bounded-memory ASM* M (of vocabulary Υ) is a triple $(\Upsilon, \mathit{initial}, \Pi)$ where *initial* is an HF-initialization mapping over Υ (i.e., for every $\mathcal{I} \in \text{Fin}(\Upsilon_{\text{in}})$, $\mathit{initial}(\mathcal{I})|_{\text{in,stat}} = \mathcal{I}^+$) and Π is a bounded-memory program over Υ. □

The program-formation rule for guarded distributed composition will become important only at the end of this section. Guarded distributed execution is an important feature of our ASM model for it allows bounded-memory ASMs to detect and terminate unproductive distributed computations (see the discussion prior to Lemma 5.3.5 and also Theorem 5.4.1).

Semantics of Bounded-Memory ASMs. Let us first consider bounded-memory ASMs whose program can be derived without the rule for guarded distributed composition. Suppose that M is such an ASM with program $\Pi = (\Pi_1, \ldots, \Pi_q)$. Informally, the semantics of Π is the sequential execution of its strata. Stratum Π_i starts on the halting state of stratum Π_{i-1} and uses, aside from the input structure, the output relations of all previous strata as input. Suppose that the halting state of Π_{i-1} is $\mathcal{S}$ and that Π_i is following distributed program:

```
do-for-all x ∈ r_x, y ∈ r_y
    Π'_{x,y}
```

where $\Pi'_{x,y}$ was obtained from some nullary program Π' by replacing every occurrence of a nullary dynamic function symbol v with $v(x, y)$. (Frequently, we will write $v_{x,y}$ instead of $v(x, y)$ to indicate that the function symbol v originated from a nullary function symbol.) The semantics of Π_i is the parallel execution of instances of $\Pi'_{x,y}$, where there is an instance $\Pi'_{a,b}$ for each pair $(a, b) \in r_x^{\mathcal{S}} \times r_y^{\mathcal{S}}$. That is, if $\Pi'_{a,b}$ denotes the 'nullary' program obtained from $\Pi'_{x,y}$ by replacing x and y with a and b, respectively, then one step of Π_i can be thought of as one step of $||_{a,b}\Pi'_{a,b}$. There is no interference between different instances of $\Pi'_{x,y}$ because each instance $\Pi'_{a,b}$ got its 'nullary' dynamic functions tagged with a, b. Π_i halts when all instances halt.

We come to formal definition of the semantics of arbitrary bounded-memory ASMs. Consider a bounded-memory ASM $M = (\Upsilon, \mathit{initial}, \Pi)$ with program $\Pi = (\Pi_1, \ldots, \Pi_q)$. Let $\mathcal{S}$ be a state over Υ. A *successor state* of $\mathcal{S}$ with respect to a stratum Π_i is defined as usual. The only new case is when Π_i was obtained by means of an application of the rule for guarded distributed composition. In that case let

$$\mathrm{Den}\left(\begin{array}{l}\texttt{do-for-all } \bar{x} \in \bar{r} \\ \texttt{unless } \varphi \\ \quad \Pi_{\bar{x}}\end{array}, \mathcal{S}\right) \;:=\; \mathrm{Den}\left(\begin{array}{l}\texttt{do-for-all } \bar{x} \in \bar{r} \\ \quad \texttt{if } \neg\varphi \texttt{ then} \\ \quad\quad \Pi_{\bar{x}}\end{array}, \mathcal{S}\right)$$

Notice that there exists indeed only one successor state of $\mathcal{S}$ with respect to Π_i because each stratum Π_i is deterministic.

As usual, an *input* $\mathcal{I}$ appropriate for M is an input over Υ. The *run* ρ of M on $\mathcal{I}$ is defined as follows. For simplicity, suppose that Π has only two strata, say, $\Pi = (\Pi_1, \Pi_2)$. Let $(\mathcal{S}_i^1)_{i\in\omega}$ be the run of Π_1 on $\mathcal{I}$, i.e., $\mathcal{S}_0^1 = \mathit{initial}(\mathcal{I})$ and for every $i \in \omega$, $\mathcal{S}_{i+1}^1$ is the successor state of $\mathcal{S}_i^1$ with respect to Π_1. If there is

no $j \in \omega$ with $\mathcal{S}^1_j = \mathcal{S}^1_{j+1}$, then let $\rho := (\mathcal{S}^1_i)_{i\in\omega}$. Otherwise, let k be the minimal j with $\mathcal{S}^1_j = \mathcal{S}^1_{j+1}$, and let $\rho := (\mathcal{S}^1_0, \dots, \mathcal{S}^1_{k-1}, \mathcal{S}^2_0, \mathcal{S}^2_1, \mathcal{S}^2_2, \dots)$, where $\mathcal{S}^2_0 := \mathcal{S}^1_k$ and for every $i \in \omega$, $\mathcal{S}^2_{i+1}$ is the successor state of $\mathcal{S}^2_i$ with respect to Π_2. This generalizes to the case where Π has more than two strata in the obvious way. The output of M on $\mathcal{I}$ is defined as for nullary ASMs.

Strictly speaking, bounded-memory ASMs whose program consists of more than one stratum are not ASMs because the ASM framework does not provide a program-formation rule for sequential composition of ASM programs. However, one can convert every bounded-memory program into an equivalent ASM program by enforcing a sequential execution of its strata. This justifies to view bounded-memory ASMs as HF-initialized ASMs. We will do so from now on.

Lemma 5.3.2. *The class of bounded-memory ASMs is closed under sequential composition in the following sense. If M_1 and M_2 are bounded-memory ASMs such that the output vocabulary of M_1 is equal to the input vocabulary of M_2, then there exists a bounded-memory ASM that simulates $(M_1; M_2)$, i.e., the sequential execution of M_1 and M_2.*

The proof of this lemma is fairly technical and is omitted here. Note that in general it does not suffice to simply concatenate the programs of M_1 and M_2 in order to obtain the program of a machine that simulates $(M_1; M_2)$. The difficulty here is that some of the elements in the output universe of M_1 may be sets. Thus, one has to conceal from M_2 the fact that certain elements in its input universe are actually not atomic.

Reductions via Bounded-Memory ASMs

Bounded-memory ASMs can serve as a basis for a reduction theory among structures. The vast majority of reductions in complexity theory are logspace reductions (see, e.g., [Pap94, GHR95]) and many computational problems considered there have input instances that are typically viewed as structures, such as circuits, graphs, networks, games, etc. We think that bounded-memory ASMs have clear advantages over Turing machines and (FO+DTC) interpretations:

1. Writing a bounded-memory program that performs the desired reduction already establishes logspace computability of the reduction. There is no need to analyze the storage requirements of a Turing machine program.

2. In many reductions, when mapping an instance of one problem to an instance of another, the image instance grows, i.e., one has to invent new elements. Bounded-memory ASMs introduce new elements by forming sets.

3. The syntax and semantics of bounded-memory programs are easy to understand—usually quantifier-free guards suffices—and match the intuition of reductions as computations.

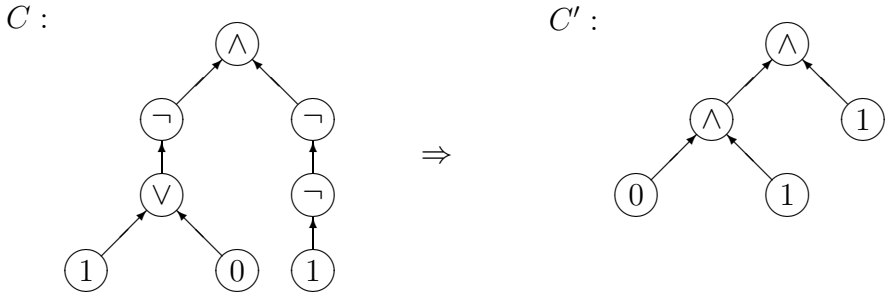
The reader can easily check the last point in the course of following sample reduction.

Example 5.3.3. Consider the PTIME-complete *circuit value problem* [Pap94]:

CV : Given a boolean circuit C and a truth assignment T on the input gates of C, decide whether C on T outputs true.

A circuit C can be viewed as a finite directed graph whose nodes represent gates and whose edges represent wires. Formally, C is a triple (V, E, Type) consisting of a finite set V, an edge relation $E \subseteq V \times V$, and a partition $\mathit{Type} = (\mathit{And}, \mathit{Or}, \mathit{Not}, \mathit{In})$ of V (determining the type of each gate in V). An instance of CV is thus a finite structure (C, T), where C is a circuit and $T \subseteq \mathit{In}$ is a truth assignment on the input gates of C.

The *monotone circuit value problem* (MCV) is the restriction of CV to circuits without *Not* gates. (Note that any boolean function computed by a circuit without *Not* gates is monotone.) CV reduces to MCV as follows. Consider a circuit that is a tree (whose root represents the output gate and whose leafs represent the input gates of the circuit). By successively applying De Morgan's Laws, we can move *Not* gates downwards to the input gates and alter the type or the truth value of the gates passed accordingly:



This reduction is not FO definable for one has to flip the truth value of an input gate iff there is an odd number of *Not* gates on the path from the input gate to the root. (Note however that there exists a slightly more complicated FO definable reduction from CV to MCV [GHR95].)

Below, we present a bounded-memory ASM M_{CV} computing this reduction. On input of an instance (C, T) of CV, M_{CV} computes an instance (C', T') of MCV such that $(C', T') \in \text{MCV}$ iff $(C, T) \in \text{CV}$. For simplicity, suppose that C is a tree whose root is not a *Not* gate. For every gate $x \in V$, M_{CV} runs a nullary program Π_x which first pebbles the gate $\mathit{succ}'(x)$ with a private nullary dynamic function pebble_x (where $\mathit{succ}'(x)$ is defined as in Example 5.2.3), and then moves pebble_x along the edges up to the root. In the meanwhile, Π_x stores the parity of the number of *Not* gates passed on the way up in a private boolean relation even_x.

If Π_x hits the root and $even_x = false$, then it switches the type or the truth value of x. We display only the program of M_{CV}; its vocabulary and initialization mapping are defined accordingly.

program Π_{CV}:

do-for-all $x \in V$

if $mode_x = initializePebble$ **then**
$pebble_x := succ'(x)$
$even_x := true$
$mode_x := movePebble$

if $mode_x = movePebble$ **then**
if $pebble \neq \varnothing$ **then**
if $Not(pebble_x)$ **then** $even_x := flip(even_x)$
$pebble_x := succ'(pebble_x)$ (1)
else
$mode_x := output$

if $mode_x = output \wedge \neg Not(x)$ **then**
$Universe(x)$ (2)
if $In(x)$ **then** $In'(x)$
if $even_x = false$ **then**
if $\neg T(x)$ **then** $T'(x)$
if $And(x)$ **then** $Or'(x)$
if $Or(x)$ **then** $And'(x)$
else
if $T(x)$ **then** $T'(x)$
if $And(x)$ **then** $And'(x)$
if $Or(x)$ **then** $Or'(x)$

The above program still does not compute the edges of the output circuit, i.e., the relation E'. A simple modification takes care of this. Suppose that $reminder_x$ denotes the first gate y on the path from x to the root such that y is not a Not gate. E' is correctly defined if we insert after line (2) the rule

if $x \neq root$ **then** $E'(x, reminder_x)$

Since on its way up, $pebble_x$ will be passing $reminder_x$ anyway, we can add the following rule after line (1) in order to initialize $reminder_x$:

if $\neg Not(pebble_x) \wedge reminder_x = \varnothing$ **then** $reminder_x := pebble_x$

□

Computational Power of Bounded-Memory ASMs

Bounded-memory ASMs compute, like nullary ASMs, partial functions from finite structures to finite structures.

Theorem 5.3.4. *Every function computable by a bounded-memory ASM is* LOGSPACE*-computable.*

The proof of this theorem is postponed until the next chapter (see below the proof of Theorem 6.1.3). It is straightforward for bounded-memory ASMs whose program does not contain guarded distributed strata. In fact, such ASMs can be simulated by Turing machines similar to the logspace-bounded Turing machine N described at the beginning of this section. The situation becomes more complicated in the presence of guarded distributed strata. To see the problem, consider the following guarded distributed program:

```
program Π:
  do-for-all x ∈ atoms
  unless halt
     Π_x
```

N would enumerate all atoms in the input structure of Π, say, in the order $a_1, \ldots, a_k$, and then simulate $\Pi_{a_1}, \ldots, \Pi_{a_k}$ in that order. Now suppose that Π_{a_1} outputs a tuple $\bar{b}$ after, say, 42 steps, and that Π_{a_k} outputs *halt* right in the first step. According to the semantics of Π all instance of Π_x are terminated after the first step. In particular, Π_{a_1} does *not* output $\bar{b}$, although N would.

In the introduction to this chapter we formulated three necessary conditions for reduction among structures. The third condition was that reductions should have at least the power of (FO+DTC) interpretations. This raises the question whether any (FO+DTC) definable query can be computed by a bounded-memory ASM. Next, we answer this question in the affirmative. (Notice that by Lemma 5.2.4, bounded-memory ASMs compute precisely the class of LOGSPACE-computable boolean queries on ordered structures. Thus, we already know that any (FO+DTC) definable query on ordered structures can be computed by a bounded-memory ASM.)

Recall Example 5.2.3 and the subsequent discussion concerning the two shortcomings of nullary ASMs on unordered structures, namely the problems of *termination* and *systematic exploration* of the input. We already fixed the second problem by adding distributed execution to nullary programs. For example, the following distributed program Π'_{DTC} (resembling Π_{DTC} in Example 5.2.3) computes $DTC(E)$ on input of an unordered graph (V, E), although it may not terminate.

```
program Π'_DTC:
  do-for-all x ∈ atoms
```

```
if mode_x = initializePebble then
    pebble_x := x
    mode_x := movePebble

if mode_x = movePebble ∧ pebble_x ≠ ∅ then
    DTC(x, pebble_x)
    pebble_x := succ'(pebble_x)
```

Observe that an instance of the nullary body of Π'_{DTC} may run into a cycle, thereby preventing Π'_{DTC} from halting. It is still not clear how to ensure termination without counting configurations.

Let us modify Π'_{DTC} a little bit. The resulting distributed program Π^*_{DTC} (see below) uses guarded distributed execution to detect and terminate every instance of the nullary body of Π'_{DTC} that 'hangs' in a cycle. In particular, Π^*_{DTC} halts on all inputs. Let *Cycle* be a new unary output symbol.

program Π^*_{DTC}:

```
do-for-all x ∈ atoms, y ∈ atoms
unless Cycle(x)

    if mode_{x,y} = initializePebble then
        pebble_{x,y} := x
        mode_{x,y} := movePebble

    if mode_{x,y} = movePebble ∧ pebble_{x,y} ≠ ∅ then
        DTC(x, pebble_{x,y})
        pebble_{x,y} := succ'(pebble_{x,y})                    (1)

    if pebble_{x,y} = y then
        if reached_{x,y} = false then
            reached_{x,y} := true
        else
            Cycle(x)
```

Let Π_{xy} denote the nullary body of Π^*_{DTC}. The new guard '`unless` $Cycle(x)$' for Π_{xy} in Π^*_{DTC} guarantees that only those instances of Π_{xy} contribute in the next computation step, for which $Cycle(x)$ does *not* hold. All other instances of Π_{xy} are disabled. Here is the idea behind Π^*_{DTC}. Fix a node $a \in V$ and concentrate on the deterministic E-path starting at a. We run $\Pi_{a,b}$ for every node $b \in V$ in parallel. Each b can be thought of as a probe. When $pebble_{a,b}$ is placed on b the first time, we set a dynamic function $reached_{a,b}$ to *true*, indicating that b has been examined once. If b is pebbled a second time, we know that the deterministic path starting at a leads into a cycle through b. In that case, there will be no further new output to $DTC(a, y)$. Thus, we can stop all $\Pi_{a,c}$ whose first subscript a is the same as that of $\Pi_{a,b}$, which detected the cycle. As a stop

signal for each $\Pi_{a,c}$, we set *Cycle*(a) to *true*. Now consider the case where there is no b such that $\Pi_{a,b}$ places $pebble_{a,b}$ on b twice. In that case, the deterministic path starting at a does not lead into a cycle. All $\Pi_{a,c}$ will simultaneously come to a halt when the path ends.

The outlined cycle-detection technique plays an important role in the proof of the following lemma as well as in the next section, where we present a construction to ensure termination of any bounded-memory ASM.

Lemma 5.3.5. *Every query definable in deterministic transitive-closure logic* (FO+DTC) *is computable by a bounded-memory ASM.*

Proof. Consider a k-ary query Q on $\mathrm{Fin}(\Upsilon)$ and suppose that $\varphi(x_1, \ldots, x_k) \in (\mathrm{FO{+}DTC})(\Upsilon)$ defines Q. For simplicity, let us assume that Υ does not contain function symbols. Let R_φ be a k-ary relation symbol not in Υ. We are going to present a bounded-memory ASM $M_{\varphi,\Upsilon}$ with input vocabulary Υ and output vocabulary $\{R_\varphi\}$ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, $M_{\varphi,\Upsilon}$ on input $\mathcal{A}$ outputs $(A, Q^{\mathcal{A}})$. To this end, define an auxiliary bounded-memory program Π_φ by induction on the construction of $\varphi(\bar{x})$. If $\varphi(\bar{x})$ is an atomic FO formula, then let Π_φ be

```
do-for-all x̄ ∈ atoms
    if φ(x̄) then R_φ(x̄)
```

Suppose that $\varphi(\bar{x}) = \exists y \psi(\bar{x}, y)$. Let Π_ψ be obtained from $\psi(\bar{x}, y)$ by induction hypothesis. Set $\Pi_\varphi = (\Pi_\psi, \Pi)$ where Π is

```
do-for-all x̄ ∈ atoms
    if (∃y ∈ atoms)R_ψ(x̄, y) then R_φ(x̄)
```

The cases $\varphi(\bar{x}) = \neg\psi(\bar{x})$ and $\varphi(\bar{x}) = \psi(\bar{y}) \vee \chi(\bar{z})$ can be treated similarly. Finally, suppose that $\varphi(\bar{x}) = [\mathrm{DTC}_{\bar{y},\bar{y}'}\ \psi(\bar{y}, \bar{y}')](\bar{t}, \bar{t}')$. We may assume that $\mathrm{free}(\psi) = \{\bar{y}, \bar{y}'\}$. (Otherwise, replace φ with an equivalent DTC-formula satisfying this condition.) Let Π_ψ be obtained from $\psi(\bar{y}, \bar{y}')$ by induction hypothesis, and let n be the length of $\bar{y}$ (and thus of $\bar{y}'$). Obtain Π^*_ψ from the distributed program Π^*_{DTC} displayed above by means of the following modifications:

1. Replace every occurrence of x (resp. y) with $\bar{y}$ (resp. $\bar{y}'$). Replace every occurrence of *Cycle* with a new n-ary output symbol $Cycle_\psi$, and every occurrence of DTC with a new $2n$-ary output symbol DTC_ψ.

2. In the obtained program, except of line (1), $pebble_{\bar{y},\bar{y}'}$ stands for the n-tuple $(pebble^1_{\bar{y},\bar{y}'}, \ldots, pebble^n_{\bar{y},\bar{y}'})$ of dynamic function symbols. For example, $pebble_{\bar{y},\bar{y}'} := \bar{y}$ now stands for n updates of the form $pebble^i_{\bar{y},\bar{y}'} := y_i$.

3. Replace line (1) with n updates of the form $pebble^i_{\bar{y},\bar{y}'} := succ^i_\psi(pebble_{\bar{y},\bar{y}'})$, where

$$succ^i_\psi(\bar{y}) \quad := \quad proj^i\big(unique\{tuple(\bar{y}') : \bar{y}' \in atoms : R_\psi(\bar{y}, \bar{y}')\}\big),$$

tuple is a (FO+BS) term denoting a function that maps n atoms to the n-tuple consisting of these atoms, and each $proj^i$ is a (FO+BS) term denoting a function that maps every n-tuple of atoms to the i-th atom in that tuple. (Here, we assume that n-tuples are defined as nested ordered pairs. Ordered pairs can be defined according to the standard Kuratowski definition; see, e.g., [BGS99, Section 6].)

Set $\Pi_\varphi = (\Pi_\psi, \Pi^*_\psi, \Pi)$ where Π is

```
do-for-all x̄, ȳ, ȳ′ ∈ atoms
    if DTC_ψ(t̄, t̄′) then R_φ(x̄)
```

To the definition of $M_{\varphi,\Upsilon}$. Choose the ASM vocabulary Υ' so that $\Upsilon'_{\text{in}} = \Upsilon$, $\Upsilon'_{\text{out}} = \{R_\varphi\}$, and Π_φ is a bounded-memory program over Υ'. Let Π be the program

```
do-for-all x ∈ atoms
    Universe(x)
```

and set $M_{\varphi,\Upsilon} = (\Upsilon', (\Pi_\varphi, \Pi))$. □

Corollary 5.3.6. *On unordered input structures, bounded-memory ASMs are more powerful than nullary ASMs.*

This follows from the existence of simple FO definable queries on unordered structures that cannot be computed by any nullary ASM (recall our discussion following Example 5.2.3). We conclude our investigation of the computational power of bounded-memory ASMs with a remark on a subtlety of guarded distributed execution.

Remark 5.3.7. Recall the program-formation rule for guarded distributed composition in Definition 5.3.1. In particular, recall that this rule is only applicable if the atomic formulas $\alpha_1, \ldots, \alpha_n$ and the nullary program Π satisfy the following condition: for every update $R(t_1, \ldots, t_k)$ of an output relation R in Π, if some α_i has the form $R(y_1, \ldots, y_k)$, then $t_i = y_i$ for every $i \in \{1, \ldots, k\}$. Intuitively, this condition limits communication between parallel running instances of $\Pi_{\bar{x}}$. For example, if we would allow an instance $\Pi_{a,b}$ to output $R(b, a)$, then this output would trigger termination of instance $\Pi_{b,a}$. We will now demonstrate that without this condition, bounded-memory ASMs would be able to compute PTIME-complete problems. This will show that, unless LOGSPACE = PTIME, limiting communication between parallel running instances of nullary programs is crucial for the LOGSPACE computability of bounded-memory ASMs.

We describe a somewhat complicated algorithm for deciding the PTIME-complete *nand circuit value problem* (NCV) [GHR95]. Suppose that we are given a circuit $C = (V, E, \mathit{Type})$ as in Example 5.3.3, except that now C has only two types of gates, namely *In* and *NAnd* gates, and a truth assignment $T \subseteq \mathit{In}$ on

the input gates of C. Our task is to extend T to a truth assignment $T' \subseteq V$ on all gates of C according to the logic of the circuit. For simplicity, we again assume that (V, E) is a tree as in Example 5.3.3. Furthermore, we assume that there is a function $\mathit{base} : V \to \mathit{In}$ such that for every gate $x \in V$:

- there is a path p from $\mathit{base}(x)$ to x, and
- the length of p is maximal in the sense that there is no $y \in \mathit{In}$ such that the path from y to x is longer than p.

It is not hard to see that *base* can be obtained from (V, E) in LOGSPACE.

Associate with each gate x two pebbles, $\mathit{pebble}_{x,\mathit{true}}$ and $\mathit{pebble}_{x,\mathit{false}}$. Initially, $\mathit{pebble}_{x,\mathit{true}}$ and $\mathit{pebble}_{x,\mathit{false}}$ are placed on $\mathit{base}(x)$. In every step, each pebble is either removed from the circuit or, if it is currently placed on some gate y, it is moved upward to $\mathit{succ}'(y)$, i.e., the successor gate of y. (succ' is defined as in Example 5.2.3.) The goal of $\mathit{pebble}_{x,\mathit{true}}$ is to reach gate $\mathit{succ}'(x)$. If it succeeds, there is an output $T'(x)$, which means that gate x evaluates to *true*. Competing with $\mathit{pebble}_{x,\mathit{true}}$, $\mathit{pebble}_{x,\mathit{false}}$ is heading for gate x. If it succeeds, the three pebbles $\mathit{pebble}_{x,\mathit{false}}$, $\mathit{pebble}_{x,\mathit{true}}$, and $\mathit{pebble}_{\mathit{succ}'(x),\mathit{false}}$ are removed from the circuit. It is important to note that removing $\mathit{pebble}_{x,\mathit{true}}$ and $\mathit{pebble}_{\mathit{succ}'(x),\mathit{false}}$ implies that:

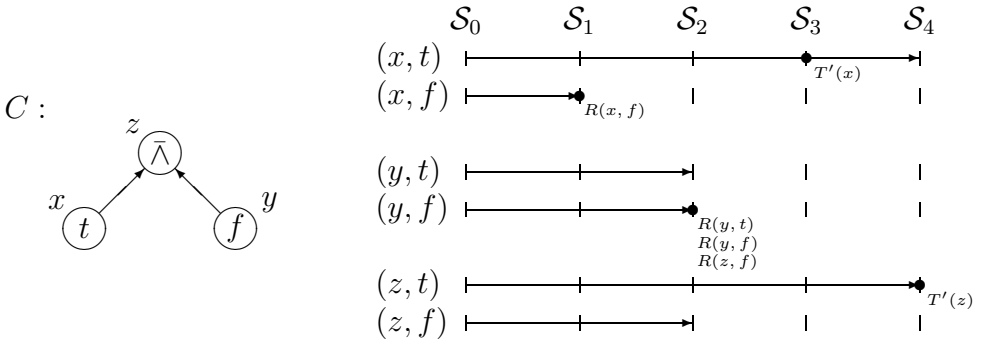
- x evaluates to *false*, because $\mathit{pebble}_{x,\mathit{true}}$ will not cause an output $T'(x)$, and
- $\mathit{succ}'(x)$ evaluates to *true*, because $\mathit{pebble}_{\mathit{succ}'(x),\mathit{true}}$ will eventually hit its target and output $T'(\mathit{succ}'(x))$.

Notice also that this reflects the logic of the *NAnd* gate $\mathit{succ}'(x)$: if x and y are the two predecessors of $\mathit{succ}'(x)$, then $\mathit{succ}'(x)$ is *true* iff x, y, or both x and y are *false*. The following bounded-memory program computes T' on input (C, T, base). It can be derived by means of a relaxed version of the rule for guarded distributed composition.

```
program Π_NCV:

  do-for-all x ∈ G, y ∈ {true, false}
  unless Remove(x, y)

     if mode_{x,y} = initializePebble then
        if T(x) then Remove(x, false)
        pebble_{x,y} := base(x)
        mode_{x,y} := movePebble

     if mode_{x,y} = movePebble ∧ y = true then
        if pebble_{x,y} ≠ succ'(x) then
           pebble_{x,y} := succ'(pebble_{x,y})
        else
           T'(x)
```

```
if mode_{x,y} = movePebble ∧ y = false then
    if pebble_{x,y} ≠ x then
        pebble_{x,y} := succ'(pebble_{x,y})
    else
        Remove(x, true)
        Remove(x, false)
        Remove(succ'(x), false)
```

As an example, consider the instance (C, T) of NCV displayed on the left-hand side of the figure below. The computation of Π_{NCV} on (C, T, base) is sketched in the diagram on the right-hand side. In the diagram, an arrow associated with a pair (x, y) represents the lifetime of $\mathit{pebble}_{x,y}$. According to the logic of C, Π_{NCV} outputs $T'(x)$ and $T'(z)$, but not $T'(y)$.



□

5.4 Choiceless Logarithmic Space

In this section, we define a complexity class called *Choiceless Logarithmic Space* (denoted $\tilde{\text{C}}$LOGSPACE) as a collection of computational problems decidable by means of (standardized) bounded-memory ASMs. $\tilde{\text{C}}$LOGSPACE can be viewed as the logarithmic-space counterpart of *Choiceless Polynomial Time* (denoted $\tilde{\text{C}}$PTIME), a complexity class that has recently been defined by Blass, Gurevich, and Shelah using *PTime bounded ASMs* [BGS99]. (A PTime bounded ASM can be viewed as a polynomial-time bounded HF-initialized ASM.) Both bounded-memory ASMs and PTime bounded ASMs are choiceless in the sense that they can form and handle sets of objects (like nodes in an input graph) without ever actually choosing one particular element of a set. Essentially this capability makes bounded-memory ASMs more expressive than the logic (FO+DTC) (recall the proof of Lemma 5.2.5), and PTime bounded ASMs more expressive than the logic (FO+LFP) (see [BGS99, Section 7]).

Unlike most complexity classes, $\tilde{\text{C}}\text{PTIME}$ is defined by means of three-valued machines: a PTime bounded ASM may accept or reject or neither accept nor reject a given input. Formally, $\tilde{\text{C}}\text{PTIME}$ is defined as a collection of pairs (C_1, C_2) where C_1 and C_2 are disjoint, but not necessarily complementary classes of finite structures over the same vocabulary. By definition, a pair (C_1, C_2) is in $\tilde{\text{C}}\text{PTIME}$ if there exists a PTime bounded ASM M that accepts all structures in C_1 and rejects all structures in C_2. On structures neither in C_1 nor in C_2, M may act arbitrarily. The definition of $\tilde{\text{C}}\text{PTIME}$ is based on three-valued machines because there seems to be no easy way of telling whether a given PTime bounded ASM qualifies as an *acceptor*, i.e., a machine which on every input either accepts or rejects the input (see also the remark in [BGS99, Section 10]).

In contrast, we show here that every bounded-memory ASM can be altered by a syntactic manipulation so that it halts on all inputs and computes exactly the same output as the original ASM, whenever the latter halts. We call bounded-memory ASMs altered this way *standard bounded-memory ASMs*. The existence of a standard form for bounded-memory ASMs allows us to define $\tilde{\text{C}}\text{LOGSPACE}$ as an ordinary (i.e., two-valued) complexity class. In order to check whether a given bounded-memory ASM qualifies as an acceptor it suffices to check whether it is a standard bounded-memory ASM equipped with a boolean output symbol *accept*.

Termination of Bounded-Memory ASMs

We prove the following theorem that generalizes Lemma 5.2.6.

Theorem 5.4.1. *Every bounded-memory ASM M can be altered by a syntactic manipulation so that the resulting bounded-memory ASM M' has the same input and output vocabulary as M, halts on every input, and produces the same output as M on every input on which M halts.*

The main idea in the proof of this theorem is to lift the cycle-detection construction leading from Π'_{DTC} to Π^*_{DTC} in the previous section to arbitrary distributed programs. We will explain the idea in more detail below. Along the way, we will develop a useful technique for decomposing hereditarily finite sets. This technique will be important in the next chapter. The actual proof of Theorem 5.4.1 is presented at the end of this subsection.

Consider the following distributed program Π obtained from a nullary program Π' by means of the rule for guarded distributed composition:

```
program Π:
  do-for-all x̄ ∈ r̄
  unless φ
     Π'_x̄
```

Fix an enumeration $v_1, \ldots, v_d$ of the nullary dynamic function symbols occurring in Π'. For the time being, let us focus on some instance $\Pi'_{\bar{a}}$ of $\Pi'_{\bar{x}}$. The successor state of a state $(\mathcal{S}, \bar{a})$ of $\Pi'_{\bar{a}}$ is entirely determined by the values $v_1^{\mathcal{S}}(\bar{a}), \ldots, v_d^{\mathcal{S}}(\bar{a})$ and the input component $\mathcal{S}|_{\text{in}}$ (see also the proof of Lemma 5.2.4). Thus, one can detect a cycle in the computation of $\Pi'_{\bar{a}}$ as follows. Choose new variables $y_1, \ldots, y_d$ and obtain $\Pi'_{\bar{x},y_1,\ldots,y_d}$ from $\Pi'_{\bar{x}}$ by replacing every occurrence of $v_i(\bar{x})$ with $v_i(\bar{x}, y_1, \ldots, y_d)$. For all possible values $b_1, \ldots, b_d$ of $v_1(\bar{a}), \ldots, v_d(\bar{a})$ in all possible states of $\Pi'_{\bar{a}}$, run an instance $\Pi'_{\bar{a},b_1,\ldots,b_d}$ of $\Pi'_{\bar{x},y_1,\ldots,y_d}$ in parallel. If one of these instances, say, $\Pi'_{\bar{a},b_1,\ldots,b_d}$, observes twice that the current values of its private nullary dynamic functions $v_1, \ldots, v_d$ match its private values $b_1, \ldots, b_d$, then there is a cycle in the computation of $\Pi'_{\bar{a}}$. $\Pi'_{\bar{a},b_1,\ldots,b_d}$ can terminate all instances of $\Pi'_{\bar{x},y_1,\ldots,y_d}$, including itself. Here is a modified version of Π that detects repeating configurations and halts on all inputs:

```
program Π*:
  do-for-all x̄ ∈ r̄, y_1, ..., y_d ∈ t
  unless φ ∨ Cycle(x̄)
    Π'_{x̄,ȳ}
    if ⋀_{i=1}^{d} v_{i,x̄,ȳ} = y_i then
      if reached_{x̄,ȳ} = false then
        reached_{x̄,ȳ} := true
      else
        Cycle(x̄)
```

where we have to choose the range term t so that $v_1^{\mathcal{S}}(\bar{a}), \ldots, v_d^{\mathcal{S}}(\bar{a}) \in t^{\mathcal{S}}$ for all possible states $(\mathcal{S}, \bar{a})$ of $\Pi'_{\bar{a}}$, on any input.

Unfortunately, we may not be able to find such a range term t for every given distributed program Π. The problem is that, if s is the maximal size bound of Π, and I denotes the current input universe of Π, then each v_i can in general assume values in $I \cup \mathrm{HF}_s(I)$. For instance, the value of v_i may range in $\{\{a\} : a \in I\} \subseteq \mathrm{HF}_2(I)$ and it is easy to see that there is no (FO+BS) term denoting (a superset of) $\{\{a\} : a \in I\}$. To solve the problem we are going to decompose every hereditarily finite set into a *form*—its structural appearance—and *matter*—the atoms it is built from. (The two notions "form" and "matter" are borrowed from [BGS99]. Though their intuitive meaning in [BGS99] is quite similar, they are completely different under formal aspects.) Once separated, the forms and matters of all elements in $I \cup \mathrm{HF}_s(I)$ can be handled by a bounded-memory ASM.

Form-Matter Decomposition. Fix a natural number $s \geq 1$ and a finite set A of atoms, and let $\mathit{Slot} := \{1, \ldots, s+1\}$. A *form* F is an element in $\mathit{Slot} \cup \mathrm{HF}(\mathit{Slot})$ satisfying $\varnothing \notin \mathrm{TC}(F)$. A *matter* $\bar{m}$ is an s-tuple of atoms in A. For every form F and matter $\bar{m} = (m_1, \ldots, m_s)$, let $\mathit{element}(F, \bar{m}) \in A \cup \mathrm{HF}(A)$ be define

recursively:

$$element(F, \bar{m}) \;:=\; \begin{cases} m_F & \text{if } F \in \{1, \dots, s\} \\ \varnothing & \text{if } F = s+1 \\ \{element(f, \bar{m}) : f \in F\} & \text{otherwise.} \end{cases}$$

Proposition 5.4.2. *For every $a \in A \cup \mathrm{HF}_s(A)$ there exists a form F and a matter $\bar{m}$ such that $element(F, \bar{m}) = a$.*

Proof. We show a stronger assertion:

Claim. Choose a set $B \subseteq A$ of cardinality $\leq s$. There is a matter $\bar{m}$ such that for every $a \in B \cup \mathrm{HF}(B)$ there is a form F with $element(F, \bar{m}) = b$.

Proof of the claim. Fix a linear order on B, say, $b_1 < b_2 < \ldots < b_k$. For every $i \in \{1, \dots, s\}$, if $i \leq k$, set $m_i = b_i$; otherwise, set $m_i = \varnothing$. This defines $\bar{m}$. We define F by induction on the rank of a. If a is an atom, then there is an $i \in \{1, \dots, k\}$ with $m_i = a$. In that case, set $F = i$. If $a = \varnothing$, set $F = s + 1$. Otherwise, suppose that $a = \{a_1, \dots, a_n\}$ is a non-empty set. Every $a_j \in a$ is in $B \cup \mathrm{HF}(B)$ and by induction hypothesis there are forms $f_1, \dots, f_n$ such that $element(f_j, \bar{m}) = a_j$ for $j \in \{1, \dots, n\}$. Set $F = \{f_1, \dots, f_n\}$. $\Diamond$

This implies the proposition as follows. For every $a \in A \cup \mathrm{HF}_s(A)$, $|\mathrm{TC}(a) \cap A| \leq s$. Let $B := \mathrm{TC}(a) \cap A$. Obviously, $a \in \mathrm{HF}(B)$ and the claim provides F and $\bar{m}$ as desired. □

In what follows, we identify $1, 2, \dots, s+1$ with the von Neumann ordinals $\{\varnothing\}, 1 \cup \{1\}, \dots, s \cup \{s\}$, respectively. Below, we define a (FO+BS) term $comp_r$ over $\{\varnothing\}$ with $\mathrm{free}(comp_r) = \{x, y_1, \dots, y_s\}$ by induction on r. Intuitively, $comp_r$ denotes a restriction of the mapping *element*. If x is interpreted as a form F, and $\bar{y}$ is interpreted as a matter $\bar{m}$ such that $element(F, \bar{m})$ has rank $\leq r$ and size $\leq s$, then $comp_r(x, \bar{y})$ denotes $element(F, \bar{m})$.

$$comp_0(x, \bar{y}) \;:=\; \begin{cases} y_1 & \text{if } x = 1 \\ \vdots & \\ y_s & \text{if } x = s \\ \varnothing & \text{otherwise} \end{cases}$$

$$comp_{r+1}(x, \bar{y}) \;:=\; \begin{cases} comp_0(x, \bar{y}) & \text{if } x \in \{1, \dots, s+1\} \\ \{comp_r(z, \bar{y}) : z \in x\}_s & \text{otherwise.} \end{cases}$$

This definition by case can be formalized in terms of (FO+BS); for details the reader may consult [BGS99, Section 6].

Now choose an atom or a set $a \in A \cup \mathrm{HF}_s(A)$ and repeat the proof of Proposition 5.4.2 for a. This time, however, instead of constructing a form F, built

a (FO+BS) term over $\{\varnothing\}$ representing F. For example, in the induction basis construct terms $\varnothing, \{\varnothing\}_2, \{\varnothing, \{\varnothing\}_2\}_3, \ldots$ instead of forms $\varnothing, 1, 2, \ldots$, and in the induction step construct a term $\{f_1, \ldots, f_n\}_{s'}$, for some suitable s', instead of the form $\{f_1, \ldots, f_n\}$. We denote the (FO+BS) term over $\{\varnothing\}$ obtained this way by F_a. Let $\bar{m}_a \in A^s$ denote the corresponding matter. It is not difficult to show that for every finite structure $\mathcal{A}$ with universe A, $comp_s^{\mathcal{A}^+}[F_a, \bar{m}_a] = a$. Suppose that $|A| \geq s$. Since $A \cup \mathrm{HF}_s(A)$ is finite, there exists a (FO+BS) term $Forms_s$ over $\{\varnothing\}$ such that for every finite structure $\mathcal{A}$ with universe A

$$Forms_s^{\mathcal{A}^+} \quad = \quad \{F_a : a \in A \cup \mathrm{HF}_s(A)\}.$$

$Forms_s$ can be built from the constant symbol $\varnothing$ using only term-formation rule **(T2)** in Definition 5.1.4. It is important to note that the definition of $Forms_s$ does not dependent on A. Each term F_a represents the structure of a, independent of the atoms in $A \cap \mathrm{TC}(a)$.

Verify that for every finite structure $\mathcal{A}$ with universe A, where not necessarily $|A| \geq s$

$$\{comp_s^{\mathcal{A}^+}[F, \bar{m}] : F \in Forms_s^{\mathcal{A}^+}, \bar{m} \in A^s\} \quad = \quad A \cup \mathrm{HF}_s(A). \tag{5.1}$$

We have sketched the proof of the following lemma.

Lemma 5.4.3. *Fix a natural number $s \geq 1$. There exist* (FO+BS) *terms $comp_s$ and $Forms_s$ over $\{\varnothing\}$ such that for every finite structure $\mathcal{A}$ with universe A equation (5.1) holds.*

Equipped with form-matter decomposition we can now replace the problematic range term t in Π^*: replace every guard $(y_i \in t)$ with the new guard $(F_i \in Forms_s, \bar{m}_i \in atoms)$ and use $comp_s(F_i, \bar{m}_i)$ instead of y_i. The entire construction follows.

Proof of Theorem 5.4.1. Consider a bounded-memory ASM $M = (\Upsilon, \Pi)$ with $\Pi = (\Pi_1, \ldots, \Pi_q)$. W.l.o.g., we can assume that each stratum Π_i is a guarded distributed program over some vocabulary Υ^i. For each Π_i do the following. Suppose that Π_i was obtained from some nullary program Π', say

```
program Π_i:
  do-for-all x̄ ∈ r̄
  unless φ
     Π'_x̄
```

Let s be the maximal size bound of Π' and let $v_1, \ldots, v_d$ be an enumeration of the nullary dynamic function symbols occurring in Π'. Introduce d new variables $F_1, \ldots, F_d$, and for each F_i introduce an s-tuple $\bar{m}_i$ of new variables. Let $comp_s$ and $Forms_s$ be obtained according to Lemma 5.4.3. The following distributed program can be derived by means of the rule for guarded distributed composition. In the program, $\bar{y}$ abbreviates the variable sequence $F_1, \ldots, F_d, \bar{m}_1, \ldots, \bar{m}_d$.

```
program Π*_i:
  do-for-all x̄ ∈ r̄, F_1, ..., F_d ∈ Forms_s, m̄_1, ..., m̄_d ∈ atoms
  unless φ ∨ Cycle(x̄)
      Π'_{x̄,ȳ}
      if ⋀_{i=1}^{d} v_{i,x̄,ȳ} = comp_s(F_i, m̄_i) then
          if reached_{x̄,ȳ} = false then
              reached_{x̄,ȳ} := true
          else
              Cycle(x̄)
```

Π_i^* is a distributed program over $\Upsilon^i \cup \{Cycle, reached\}$, where *Cycle* is a new output relation symbol and *reached* a new dynamic function symbol. The bounded-memory ASM M' can now be defined as (Υ', Π') where $\Pi' := (\Pi_1^*, \ldots, \Pi_q^*)$ and Υ' is obtained from Υ by enriching it with the newly introduced symbols. □

The Definition of Choiceless Logarithmic Space

A *standard bounded-memory ASM* is a bounded-memory ASM altered according to Theorem 5.4.1. A *bounded-memory acceptor* is a standard bounded-memory ASM whose output vocabulary contains the boolean symbol *accept*. Acceptance and rejection of an input by a bounded-memory acceptor is defined in the obvious way.

Definition 5.4.4. A class C of finite structures over some vocabulary Υ is in *Choiceless Logarithmic Space* ($\tilde{\text{C}}$Logspace) if C is closed under isomorphisms and there exists a bounded-memory acceptor that accepts every structure in C and rejects every structure in $\text{Fin}(\Upsilon) - C$. □

The following figure summarizes our results concerning the computational power of bounded-memory ASMs in this chapter:

$$\text{(FO+DTC)} \quad \subset \quad \tilde{\text{C}}\text{Logspace} \quad \subseteq \quad \text{Logspace}$$

where (FO+DTC) stands for the class of boolean queries definable in deterministic transitive-closure logic. (Recall that every boolean query on $\text{Fin}(\Upsilon)$ can be viewed as a subclass of $\text{Fin}(\Upsilon)$ closed under isomorphisms.) The first inclusion is implied by Lemma 5.3.5. It is proper due to Lemma 5.2.5. The second inclusion follows from Theorem 5.3.4. In the next chapter, we show that $\tilde{\text{C}}$Logspace is a proper subclass of Logspace.

6

Logical Descriptions of Choiceless Computations

Descriptive complexity theory has established numerous surprising connections between the computational complexity of a problem and its *descriptive complexity*, i.e., the richness of a language needed to describe the problem. Consider, for example, a boolean query Q on finite ordered structures. It is a well-known result due to Immerman that Q is computable by a logarithmic-space bounded Turing machine iff Q is definable in deterministic transitive-closure logic (FO+DTC) [Imm87]. A similar result due to Immerman and Vardi says that Q is computable by a polynomial-time bounded Turing machine iff Q is definable in inflationary fixed-point logic (FO+IFP) [Var82, Imm86]. In fact, many such connections between resource-bounded computability and logical definability are known in descriptive complexity theory and it has become common to say that a logic L *captures* a complexity class K if the class of boolean queries definable in L coincides with the class of boolean queries computable within the complexity bounds of K. For instance, the logic (FO+DTC) (resp. (FO+IFP)) captures the complexity class LOGSPACE (resp. PTIME) on finite ordered structures. For a comprehensive introduction to descriptive complexity theory the reader is referred to [EF95, Imm98].

In this chapter, we show that suitable extensions of the logics (FO+DTC) and (FO+IFP) capture $\tilde{\text{C}}$LOGSPACE and (an extension of) $\tilde{\text{C}}$PTIME, respectively. First, we add to the logic (FO+BS) defined in Section 5.1 a bounded version of the deterministic transitive-closure operator DTC and prove that the obtained extension of (FO+DTC), denoted (FO+BS+BDTC), captures $\tilde{\text{C}}$LOGSPACE. We then remove from (FO+BS) the explicit size bounds at set terms and add a polynomially-bounded inflationary fixed-point operator. The obtained extension

of (FO+IFP), denoted (FO+HS+PIFP), is a PTIME logic in which all $\tilde{\text{C}}$PTIME problems are expressible. The following table illustrates the two classical capturing results for LOGSPACE$^<$ and PTIME$^<$ mentioned above and our results for $\tilde{\text{C}}$LOGSPACE and $\tilde{\text{C}}$PTIME:

class	computation model	logic
LOGSPACE$^<$	logarithmic-space bounded TMs	(FO+DTC)
PTIME$^<$	polynomial-time bounded TMs	(FO+IFP)
$\tilde{\text{C}}$LOGSPACE	bounded-memory ASMs	(FO+BS+BDTC)
$\tilde{\text{C}}$PTIME	PTime bounded ASMs	(FO+HS+PIFP)

For each of the complexity classes on the left-hand side the table shows a computation model defining the class and a logic suitable for describing computations of the corresponding machines. The first three classes are even captured by the corresponding logics on the right-hand side.

Aside from providing alternative characterizations of the two choiceless complexity classes $\tilde{\text{C}}$LOGSPACE and $\tilde{\text{C}}$PTIME, our logical descriptions of choiceless computations are interesting for another reason. They make $\tilde{\text{C}}$LOGSPACE and $\tilde{\text{C}}$PTIME amenable to the combinatorial and model-theoretic techniques developed by Blass, Gurevich, and Shelah in [BGS99]. In fact, using their techniques and results we can show that $\tilde{\text{C}}$LOGSPACE is a proper subclass of both LOGSPACE and $\tilde{\text{C}}$PTIME, and that (FO+HS+PIFP) does not capture PTIME.

Related Work. Our investigations in this chapter are inspired by the work of Blass, Gurevich, and Shelah in [BGS99]. This is particularly true for the logic (FO+HS+PIFP), an outline of which is already present in their First Fixed-Point Theorem.

Outline of the Chapter. The chapter consists of two sections. In the first section, we introduce the logic (FO+BS+BDTC), show that it captures $\tilde{\text{C}}$LOGSPACE, and separate $\tilde{\text{C}}$LOGSPACE from LOGSPACE and $\tilde{\text{C}}$PTIME. In the last section, we present the logic (FO+HS+PIFP) and prove that it is a PTIME logic in which all $\tilde{\text{C}}$PTIME problems are expressible but which does not capture PTIME.

6.1 A Logic for Choiceless Logarithmic Space

Recall the logic (FO+BS) from Section 5.1. We add to (FO+BS) a bounded version of the deterministic transitive-closure operator DTC.

Definition 6.1.1. The logic (FO+BS+BDTC) is obtained from (FO+BS) by means of the following additional formula-formation rule:

(BDTC) If φ is a formula, s is a natural number, $\bar{x}$ and $\bar{x}'$ are two k-tuples of variables such that the variables among $\bar{x}, \bar{x}'$ are pairwise distinct, and $\bar{t}$ and $\bar{t}'$ are two k-tuples of terms, then $[\mathrm{DTC}_{\bar{x},\bar{x}'}\ \varphi]_s(\bar{t},\bar{t}')$ is a formula.

Formulas of the form $[\mathrm{DTC}_{\bar{x},\bar{x}'}\ \varphi]_s(\bar{t},\bar{t}')$ are also referred to as *bounded* DTC *formulas*. The *free* and *bound variables* of (FO+BS+BDTC) formulas are defined as for (FO+DTC) formulas. □

Semantics of (FO+BS+BDTC) Formulas. The semantics of a bounded DTC formula $[\mathrm{DTC}_{\bar{x},\bar{x}'}\ \varphi]_s(\bar{t},\bar{t}')$ is similar to the semantics of the unbounded version $[\mathrm{DTC}_{\bar{x},\bar{x}'}\ \varphi](\bar{t},\bar{t}')$, except that now, in order to reach $\bar{t}'$ from $\bar{t}$ via a deterministic φ-path, one may compose this path from φ-edges connecting points in $(A \cup \mathrm{HF}_s(A))^k$ rather than A^k only. (Recall that $\mathrm{HF}_s(A)$ denotes the restriction of $\mathrm{HF}(A)$ to sets of size $\leq s$.) Formally, the semantics of a bounded DTC formula is defined as follows. Let Υ be a relational vocabulary and let $[\mathrm{DTC}_{\bar{x},\bar{x}'}\ \varphi]_s(\bar{t},\bar{t}')$ be a bounded DTC formula over Υ^+ with $\mathrm{free}(\varphi) \subseteq \{\bar{x},\bar{x}',\bar{y}\}$. W.l.o.g., we can assume that none of the variables among $\bar{x},\bar{x}'$ occurs free in $\bar{t},\bar{t}'$. Consider a finite structure $\mathcal{A}$ over Υ with universe A. Choose from the universe of $\mathcal{A}^+$ interpretations $\bar{b}$ of the variables $\bar{y}$, set $\varphi^{(\mathcal{A}^+,\bar{b})} = \{(\bar{a},\bar{a}') : \mathcal{A}^+ \models \varphi[\bar{a},\bar{a}',\bar{b}]\}$, and let $\varphi_s^{(\mathcal{A}^+,\bar{b})}$ be the restriction of $\varphi^{(\mathcal{A}^+,\bar{b})}$ to $A \cup \mathrm{HF}_s(A)$. Then

$$\mathcal{A}^+ \models [\mathrm{DTC}_{\bar{x},\bar{x}'}\ \varphi]_s(\bar{t},\bar{t}')[\bar{b}] \quad :\Leftrightarrow \quad \big(\bar{t}^{\mathcal{A}^+}[\bar{b}], \bar{t}'^{\mathcal{A}^+}[\bar{b}]\big) \in DTC\big(\varphi_s^{(\mathcal{A}^+,\bar{b})}\big),$$

where $DTC\big(\varphi_s^{(\mathcal{A}^+,\bar{b})}\big)$ denotes the deterministic transitive closure of $\varphi_s^{(\mathcal{A}^+,\bar{b})}$.

Let Υ be a relational vocabulary. Every (FO+BS+BDTC) sentence over Υ^+ defines a boolean query on $\mathrm{Fin}(\Upsilon)$ in the obvious way.

Lemma 6.1.2. *Every boolean query definable in the logic* (FO+BS+BDTC) *is* LOGSPACE*-computable.*

The proof of this lemma is analogous to the proof of Lemma 5.1.6 and is omitted here.

Theorem 6.1.3. *A boolean query is definable in the logic* (FO+BS+BDTC) *iff it is computable by a bounded-memory ASM.*

Proof. Fix a relational vocabulary Υ. For the "only if" direction consider a (FO+BS+BDTC) formula $\varphi(x_1,\ldots,x_k)$ over Υ^+ with $\mathrm{free}(\varphi) = \{x_1,\ldots,x_k\}$. Let s be the maximal size bound of φ (where we now assume that s is also an upper bound for all subscripts at bounded DTC subformulas of φ) and let R_φ be a k-ary relation symbol not in Υ. We follow the proof of Lemma 5.3.5 and

construct a bounded-memory ASM $M^s_{\varphi,\Upsilon}$ with input vocabulary Υ and output vocabulary $\{R_\varphi\}$ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, $M^s_{\varphi,\Upsilon}$ on input $\mathcal{A}$ outputs $(A, \varphi_s^{\mathcal{A}})$ where $\varphi_s^{\mathcal{A}}$ is the restriction of $\{\bar{a} : \mathcal{A}^+ \models \varphi[\bar{a}]\}$ to $A \cup \mathrm{HF}_s(A)$.

The inductive definition of the auxiliary bounded-memory program Π_φ is now slightly more complicated due to nested occurrences of set terms and bounded DTC formulas. Let us first consider the case $\varphi(\bar{x}) = [\mathrm{DTC}_{\bar{y},\bar{y}'}\ \psi(\bar{y},\bar{y}')]_q(\bar{t},\bar{t}')$. W.l.o.g., we can assume that $\mathrm{free}(\psi) = \{\bar{y},\bar{y}'\}$. Let Π_ψ be obtained from $\psi(\bar{y},\bar{y}')$ by induction hypothesis. Define Π^*_ψ as in the proof of Lemma 5.3.5, except that now $\bar{y},\bar{y}'$ range in $A \cup \mathrm{HF}_q(A)$ rather than A. This can be achieved by means of form-matter decomposition. For instance, introduce for each y_i new variables $F_i, \bar{m}_i$ and replace

- (`do-for-all` $\bar{y},\bar{y}' \in$ *atoms*) with (`do-for-all` $\bar{F},\bar{F}' \in Forms_q, \bar{m},\bar{m}' \in$ *atoms*),
- the subscripts $\bar{y},\bar{y}'$ at each function symbol $pebble^i$ with the subscripts $\bar{F},\bar{F}',\bar{m},\bar{m}'$, and
- every occurrence of the variable y_i in the body of the resulting program with the (FO+BS) term $comp_q(F_i,\bar{m}_i)$.

Set $\Pi_\varphi = (\Pi_\psi, \Pi^*_\psi, \Pi)$ where Π is defined as in the corresponding induction step in the proof of Lemma 5.3.5, except that now $\bar{x},\bar{y},\bar{y}'$ range in $A \cup \mathrm{HF}_s(A)$.

Suppose that $\varphi(\bar{x})$ is an atomic formula. If $\varphi(\bar{x})$ is a (FO+BS) formula, then let Π_φ be defined as in the proof of Lemma 5.3.5. Otherwise, $\varphi(\bar{x})$ has subterms of the form $\{t : z \in r : \chi\}$ where χ is built from atomic formulas and bounded DTC formulas by means of negation and disjunction. For simplicity, let us assume that $\varphi(\bar{x})$ has precisely one such subterm $\{t : z \in r : \chi\}$ and that $\chi = [\mathrm{DTC}_{\bar{y},\bar{y}'}\ \psi(\bar{y},\bar{y}')]_q(\bar{t},\bar{t}')$. Let Π_ψ and Π^*_ψ be obtained from ψ as in the previous induction step. Suppose that DTC_ψ is the relation symbol containing the output of Π^*_ψ. Obtain ψ' from ψ by replacing in its subterm $\{t : z \in r : \chi\}$ the formula χ with $DTC_\psi(\bar{t},\bar{t}')$. Set $\Pi_\varphi = (\Pi_\psi, \Pi^*_\psi, \Pi)$ where Π is

```
do-for-all x̄
   if ψ'(x̄) then R_φ(x̄)
```

and the variables in $\bar{x}$ range in $A \cup \mathrm{HF}_s(A)$. The cases $\varphi(\bar{x}) = \neg\psi(\bar{x})$ and $\varphi(\bar{x}) = \psi(\bar{x}) \vee \chi(\bar{x})$ can be handled as in the proof of Lemma 5.3.5. Finally, to obtain the bounded-memory ASM $M^s_{\varphi,\Upsilon}$ proceed as in the proof of Lemma 5.3.5 with the exception that in the last stratum, which defines the output universe of $M^s_{\varphi,\Upsilon}$, x ranges in $A \cup \mathrm{HF}_s(A)$.

For the "if" direction consider a bounded-memory ASM M with input vocabulary Υ and output vocabulary $\{accept\}$. We define a (FO+BS+BDTC) sentence φ_M over Υ^+ such that, if M halts on an input $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, then M accepts $\mathcal{A}$ iff $\mathcal{A}^+ \models \varphi_M$. First observe that it suffices to consider the case where the program of M has only one stratum. In all other cases, we can define the output

relations of the first stratum Π_1 by means of some (FO+BS+BDTC) formulas and then, while defining the output relations of the second stratum Π_2, replace all output symbols of Π_1 with the corresponding (FO+BS+BDTC) definitions already obtained in the first step, and so forth.

To simplify the definition of φ_M, suppose that the (sole) stratum of M is the following guarded distributed program:

```
do-for-all  x ∈ r_x, y ∈ r_y
unless  R_0 ∨ R_1(x) ∨ R_2(x, y)
    Π_{x,y}
```

where $\Pi_{x,y}$ was obtained from some nullary program Π and each R_i is an output relation symbol of arity i. Fix some enumeration $v_1, \ldots, v_d$ of the dynamic function symbols in the vocabulary of M. $v_1, \ldots, v_d$ are nullary symbols in Π. Associate with each R_i a new nullary dynamic function symbol rem_i, which will serve as a *rem*inder bit being *true* iff there has previously been some output to R_i. Obtain Π' from Π by (i) removing every update of the form $Universe(t)$, and (ii) replacing every update of the form $R_i(\bar{t})$ with $rem_i := true$. In the context of Π', we view *accept* as a nullary dynamic function symbol with range $\{true, false\}$. Thus, the input vocabulary of Π' is Υ, its dynamic vocabulary is $\{v_1, \ldots, v_d, rem_0, rem_1, rem_2, accept\}$, and its output vocabulary is empty. x and y are the only free variables of Π'. In favor of a uniform notation for the dynamic symbols of Π', we also write v_{d+1+i} instead of rem_i, and v_{d+4} instead of *accept*.

Claim. For each v_i, $i \in \{1, \ldots, d+4\}$, choose two new variables z and z'. There exists a (FO+BS) formula $\chi_\Pi(x, y, \bar{z}, \bar{z}')$ over Υ^+ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ and all interpretations $a, b, \bar{c}, \bar{c}'$ of $x, y, \bar{z}, \bar{z}'$ (chosen from the universe of $\mathcal{A}^+$)

$$\mathcal{A}^+ \models \chi_\Pi[a, b, \bar{c}, \bar{c}'] \;\Leftrightarrow\; ((\mathcal{A}^+, a, b, \bar{c}), (\mathcal{A}^+, a, b, \bar{c}')) \in Trans_\Pi,$$

where $(\mathcal{A}^+, a, b, \bar{c})$ (resp. $(\mathcal{A}^+, a, b, \bar{c}')$) denotes the state over $\Upsilon^+ \cup \{x, y, \bar{v}\}$ whose input and static component is $(\mathcal{A}^+, a, b)$ and whose dynamic component is such that each v_i is interpreted as the i-th element in $\bar{c}$ (resp. $\bar{c}'$).

The proof of the claim is similar to that of Proposition 1.2.6 and omitted here. Let $next \in (\mathrm{FO+BS})(\Upsilon^+)$ with $\mathrm{free}(next) \subseteq \{x, y, \bar{z}, \bar{z}'\}$ be defined by:

$$\begin{aligned} next(x, y, \bar{z}, \bar{z}') &:= \big(\neg reminders(\bar{z}) \wedge \chi_\Pi(x, y, \bar{z}, \bar{z}')\big) \vee \\ &\quad \big(reminders(\bar{z}) \wedge \bar{z} = \bar{z}'\big) \\ reminders(\bar{z}) &:= \bigvee_{i=0}^{2} z_{d+1+i} = true. \end{aligned}$$

Let s be the maximal size bound of Π, and let $\mathcal{A}$ be a finite structure over Υ such that during the run of M on $\mathcal{A}$ there is no output to R_0 nor R_1. (Notice that the choice of $\mathcal{A}$ ensures that the computations of all instances of $\Pi_{x,y}$ do not interfere with each other. For example, if an instance $\Pi_{a,b}$ of $\Pi_{x,y}$ would output $R_1(a)$,

then the computations of all instances $\Pi_{a',b'}$ with $a' = a$ would be terminated.) The following sentence over Υ^+ holds in $\mathcal{A}^+$ iff M accepts $\mathcal{A}$:

$$\begin{aligned} \mathit{run} \;:=\; & (\exists x \in r_x)(\exists y \in r_y)(\exists \bar{z}') \\ & \big([\mathrm{DTC}_{\bar{z},\bar{z}'}\ \mathit{next}(x,y,\bar{z},\bar{z}')]_s(\overline{\varnothing},\bar{z}') \wedge z'_{d+4} = \mathit{true}\big), \end{aligned}$$

where the variables in $\bar{z}'$ range in $A \cup \mathrm{HF}_s(A)$. Using form-matter decomposition one can define a (FO+BS+BDTC) sentence which is equivalent to *run*. For simplicity, let us pretend that *run* is this (FO+BS+BDTC) sentence.

We now take interferences of instances of $\Pi_{x,y}$ into account. Consider the following two formulas over Υ^+:

$$\begin{aligned} \mathit{parallel}(xy\bar{z}', x'y'\bar{u}') \;:=\; & [\mathrm{DTC}_{\bar{z}\bar{u},\bar{z}'\bar{u}'}\ \mathit{next}(x,y,\bar{z},\bar{z}') \wedge \\ & \qquad \mathit{next}(x',y',\bar{u},\bar{u}')]_s(\overline{\varnothing}\,\overline{\varnothing},\bar{z}'\bar{u}') \\ \mathit{disabled}(xy\bar{z}') \;:=\; & (\exists x' \in r_x)(\exists y' \in r_y)(\exists \bar{u}') \\ & \big(\mathit{parallel}(xy\bar{z}', x'y'\bar{u}') \wedge [(z_{d+1} = \mathit{true}) \vee \\ & \qquad (z_{d+2} = \mathit{true} \wedge x = x')]\big), \end{aligned}$$

where the variables in $\bar{u}'$ range in $A \cup \mathrm{HF}_s(A)$. For a better understanding of the meaning of *parallel* and *disabled*, let us first assume that both formulas speak about an interference-free, initial segment of a run of M. $\mathit{parallel}(xy\bar{z}', x'y'\bar{u}')$ expresses that, if $\Pi_{x,y}$ and $\Pi_{x',y'}$ both start in the initial state, then, after the same number of steps without interference, $\Pi_{x,y}$ reaches a state with dynamic component $\bar{z}'$, and $\Pi_{x',y'}$ reaches a state with dynamic component $\bar{u}'$. $\mathit{disabled}(xy\bar{z}')$ says that, if $\Pi_{x,y}$ reaches a state with dynamic component $\bar{z}'$, then, in the next step, it is disabled by some parallel running $\Pi_{x',y'}$ that outputs R_0 or $R_1(x)$. With the help of *disabled* we can define the one-step semantics of each instance of $\Pi_{x,y}$ in terms of (FO+BS+BDTC) such that interferences with other instances are taken into account:

$$\begin{aligned} \mathit{next}'(x,y,\bar{z},\bar{z}') \;:=\; & \big(\neg \mathit{disabled}(xy\bar{z}) \wedge \mathit{next}(x,y,\bar{z},\bar{z}')\big) \vee \\ & \big(\mathit{disabled}(xy\bar{z}) \wedge \bar{z} = \bar{z}'\big) \end{aligned}$$

The desired (FO+BS+BDTC) sentence φ_M expressing that M accepts an input can now be defined like the above sentence *run* but with *next*′ substituted for *next*. □

Theorem 6.1.3 and Lemma 6.1.2 together imply that every boolean query computable by a bounded-memory ASM is Logspace-computable. We still have to prove our more general assertion in Theorem 5.3.4, namely that every function computable by a bounded-memory ASM is Logspace-computable.

Proof of Theorem 5.3.4. Consider a bounded-memory ASM M of vocabulary Υ. Following the "if" direction in the proof of the last theorem, one can define for

every k-ary relation symbol $R \in \Upsilon_{\text{out}}$ a (FO+BS+BDTC) sentence $\varphi_{M,R}(\bar{x})$ over Υ^+_{in} such that, if M halts on an input $\mathcal{A}$, and $R^{M,\mathcal{A}}$ is the interpretation of R in the output of M on $\mathcal{A}$, then $R^{M,\mathcal{A}} = \{\bar{a} : \mathcal{A}^+ \models \varphi_{M,R}[\bar{a}]\}$. Let s be the maximal size bound of the program of M. It is easy to see that for every input $\mathcal{A}$ on which M halts, $R^{M,\mathcal{A}}$ is a relation on $A \cup \mathrm{HF}_s(A)$. By Lemma 6.1.2 and the claim in the proof of Lemma 5.1.6, the following can be done in Logspace: enumerate all k-tuples over $A \cup \mathrm{HF}_s(A)$ and decide for each such tuple $\bar{a}$ whether $\mathcal{A}^+ \models \varphi_{M,R}[\bar{a}]$. In this way, one can compute $R^{M,\mathcal{A}}$ from $\mathcal{A}$ in Logspace. We conclude that for every input $\mathcal{A}$ on which M halts, the output structure of M on $\mathcal{A}$ is Logspace-computable from $\mathcal{A}$. □

Bounded-memory ASMs vs PTime bounded ASMs

Let S be a set symbol and let C be the class of finite structures $\mathcal{A} = (A, S^{\mathcal{A}})$ where $|S^{\mathcal{A}}|! \leq |A|$. The following boolean query on C is computable by a PTime bounded ASM and is thus in $\tilde{\text{C}}$Ptime [BGS99, Theorem 21]:

small-subset-parity : Given some $\mathcal{A} \in C$, decide whether $|S^{\mathcal{A}}|$ is even.

We next show that this query is not computable by any bounded-memory ASM.

Theorem 6.1.4. small-subset-parity *is not definable in* (FO+BS+BDTC).

Proof. Let us first recall an observation and some terminology from [BGS99]. Consider a finite structure $\mathcal{A}$ over some relational vocabulary Υ and suppose that A is the universe of $\mathcal{A}$.

- There is a one-to-one correspondence between automorphisms of $\mathcal{A}$ and automorphisms of $\mathcal{A}^+$. Every automorphism θ of $\mathcal{A}$ can be extended to an automorphism θ^+ of $\mathcal{A}^+$ by the recipe $\theta^+(a) = \{\theta^+(b) : b \in a\}$. It is easy to see that there is no other automorphism θ' of $\mathcal{A}^+$ satisfying $\theta'(a) = \theta(a)$ for every $a \in A$. Furthermore, every automorphism of $\mathcal{A}^+$ is an automorphism of $\mathcal{A}$ when restricted to A.

- A set $X \subseteq A$ is called a *support* of an element a of $\mathcal{A}^+$ if every automorphism of $\mathcal{A}$ that pointwise fixes X fixes a as well. For example, $A \cap \mathrm{TC}(a)$ is a support of a.

- Let k be a natural number ≥ 1. An element a of $\mathcal{A}^+$ is called *k-symmetric* if every $b \in \mathrm{TC}(a)$ has a support of cardinality $\leq k$. Obviously, any element a of $\mathcal{A}^+$ is $|A \cap \mathrm{TC}(a)|$-symmetric. In particular, $\varnothing$ and all atoms are k-symmetric for any $k \geq 1$.

- Let $\bar{\mathcal{A}}$ denote $\mathcal{A}^+|(\Upsilon \cup \{\in, \varnothing\})$, i.e., the reduct of $\mathcal{A}^+$ to the vocabulary $\Upsilon \cup \{\in, \varnothing\}$, and let $\bar{\mathcal{A}}_k$ be the restriction of $\bar{\mathcal{A}}$ to the k-symmetric elements in $\bar{\mathcal{A}}$.

Now let S and C be as in the definition of SMALL-SUBSET-PARITY and set $\Upsilon = \{S\}$. We show that there exists no sentence $\varphi \in (\text{FO+BS+BDTC})(\Upsilon^+)$ such that for every $\mathcal{A} \in C$, $\mathcal{A}^+ \models \varphi$ iff $|S^{\mathcal{A}}|$ is even. Toward a contradiction, assume that there is such a sentence φ. Let $\mathrm{L}^m_{\infty,\omega}$ denote the m-variable fragment of finite variable infinitary logic (see, e.g., [EF95]). There exist a natural number $m > 0$ and a sentence $\bar{\varphi} \in \mathrm{L}^m_{\infty,\omega}(\{S, \in, \varnothing\})$ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, $\bar{\mathcal{A}} \models \bar{\varphi}$ iff $\mathcal{A}^+ \models \varphi$. ($\bar{\varphi}$ can be obtained from φ by unfolding DTC subformulas of φ.) Let s be the maximal size bound of φ. Verify that the following holds for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ and every transitive superset A' of $A \cup \mathrm{HF}_s(A)$: if $\bar{\mathcal{A}}'$ is the restriction of $\bar{\mathcal{A}}$ to A', then $\bar{\mathcal{A}}' \models \bar{\varphi}$ iff $\bar{\mathcal{A}} \models \bar{\varphi}$. By [BGS99, Corollary 41], we can choose a positive instance $\mathcal{A}$ and a negative instance $\mathcal{B}$ of SMALL-SUBSET-PARITY such that $\bar{\mathcal{A}}_s$ and $\bar{\mathcal{B}}_s$ are $\mathrm{L}^m_{\infty,\omega}$-equivalent. Since every $a \in A \cup \mathrm{HF}_s(A)$ is supported by $A \cap \mathrm{TC}(a)$, $A \cup \mathrm{HF}_s(A)$ is a subset of the universe of $\bar{\mathcal{A}}_s$. Since $\mathcal{A}$ is a positive instance of SMALL-SUBSET-PARITY, we have $\mathcal{A}^+ \models \varphi$ and, by the above, $\bar{\mathcal{A}}_s \models \bar{\varphi}$. A similar argument shows $\bar{\mathcal{B}}_s \not\models \bar{\varphi}$. This is a contradiction, for $\bar{\mathcal{A}}_s$ and $\bar{\mathcal{B}}_s$ are $\mathrm{L}^m_{\infty,\omega}$-equivalent. □

Corollary 6.1.5. *(1) PTime bounded ASMs are more powerful than bounded-memory ASMs. (2)* C̃LOGSPACE *is a proper subclass of* LOGSPACE.

Proof. It is not hard to show that every bounded-memory ASM can be simulated by a PTime bounded ASM. On the other hand, there exists a PTime bounded ASM that computes SMALL-SUBSET-PARITY [BGS99, Theorem 21]. This ASM cannot be simulated by any bounded-memory ASM due to Theorems 6.1.3 and 6.1.4. This shows the first assertion. The second follows from Theorems 6.1.3 and 6.1.4, Lemma 6.1.2, and the observation that SMALL-SUBSET-PARITY is LOGSPACE-computable. □

Let C̃LOGSPACE$^+$ denote the extension of C̃LOGSPACE to a three-valued complexity class like C̃PTIME. That is, C̃LOGSPACE$^+$ is a collection of pairs (C_1, C_2) where C_1 and C_2 are sets of finite structures over the same vocabulary. A pair (C_1, C_2) is in C̃LOGSPACE$^+$ iff there exists a bounded-memory ASM that accepts all structures in C_1 and rejects all structures in C_2. The last proof immediately implies the following corollary.

Corollary 6.1.6. C̃LOGSPACE$^+$ *is a proper subclass of* C̃PTIME.

6.2 A Logic for Choiceless Polynomial Time

In this last section, we present a PTIME logic in which all C̃PTIME problems are expressible. The logic is obtained from (FO+BS) by removing the explicit size bounds at set terms and by adding a polynomially-bounded inflationary fixed-point operator.

Definition 6.2.1. *Terms* and *formulas* of the logic (FO+HS) are defined by simultaneous induction:

(T1') Every variable is a term.

(T2') The constant symbols $\varnothing$ and *atoms* are terms, and if t is a term, then $\mathit{unique}(t)$ and $\bigcup t$ are also terms.

(T3') If $t_1, \ldots, t_k$ are terms, then $\{t_1, \ldots, t_k\}$ is a term.

(T4') If t is a term, x is a variable, φ is a formula, and r is a term without free occurrences of x, then $\{t : x \in r : \varphi\}$ is a term.

(F1) Every atomic formula (built from terms as usual) is a formula.

(F2) If φ and ψ are formulas, then $\varphi \vee \psi$ and $\neg\varphi$ are formulas.

The logic (FO+HS+PIFP) is obtained from (FO+HS) by means of the following additional term-formation rule:

(PIFP) If t is a term, x is a variable, and $p(n)$ is a polynomial, then $[\mathrm{IFP}_x\, t]_p$ is a term.

Terms of the form $[\mathrm{IFP}_x\, t]_p$ are also called *polynomially-bounded* IFP *terms*. The *free* and *bound variables* of (FO+HS+PIFP) terms and formulas are defined in the obvious way. In particular, a variable occurs free in $[\mathrm{IFP}_x\, t]_p$ if it occurs free in t and is different from x; x itself occurs bound in $[\mathrm{IFP}_x\, t]_p$. □

Semantics of (FO+HS+PIFP) Formulas. The semantics of (FO+HS) terms and formulas is defined similar to the semantics of (FO+BS) terms and formulas, except that now we do not impose upper bounds on the size of definable sets. It remains to define the semantics of polynomially-bounded IFP terms. Let Υ be a relational vocabulary and let $[\mathrm{IFP}_x\, t]_p$ be a polynomially-bounded IFP term over Υ^+ with $\mathrm{free}(t) \subseteq \{x, \bar{y}\}$. Consider a finite structure $\mathcal{A}$ over Υ with universe A. Choose from the universe of $\mathcal{A}^+$ interpretations $\bar{b}$ of the variables $\bar{y}$ and define a sequence $(X_i)_{i\in\omega}$ of sets by induction on i:

$$\begin{aligned} X_0 &:= \varnothing \\ X_{i+1} &:= X_i \cup t^{\mathcal{A}^+}[X_i, \bar{b}] \end{aligned}$$

Let n be the cardinality of A. If there exists a $k \in \omega$ with $X_k = X_{k+1} \in \mathrm{HF}_{p(n)}(A)$, then set $[\mathrm{IFP}_x t]_p^{\mathcal{A}^+}[\bar{b}] = X_k$; otherwise, set $[\mathrm{IFP}_x t]_p^{\mathcal{A}^+}[\bar{b}] = \varnothing$. Note that, if k exists, then it is uniquely determined and $\leq p(n)$.

Lemma 6.2.2. *Let Υ be a relational vocabulary. For every* (FO+BS+BDTC) *sentence φ over Υ^+ there exists a* (FO+HS+PIFP) *sentence φ' over Υ^+ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, $\mathcal{A}^+ \models \varphi \leftrightarrow \varphi'$.*

The idea in the proof of this lemma is to replace in a given (FO+BS+BDTC) formula all occurrences of (FO+BS) set terms and bounded DTC formulas with 'equivalent' (FO+HS+PIFP) terms and formulas. For instance, one can replace an occurrence of $\{t : x \in r : \psi\}_s$ with $[\mathrm{IFP}_y \{t : x \in r : \psi\}]_s$ where y is a new variable. Using form-matter decomposition and polynomially-bounded IFP terms one can replace every occurrence of a bounded DTC formula with an appropriate (FO+HS+PIFP) formula. We omit the details.

Let Υ be a relational vocabulary. Every (FO+HS+PIFP) sentence over Υ^+ defines a boolean query on $\mathrm{Fin}(\Upsilon)$ in the obvious way.

Lemma 6.2.3. *Every boolean query definable in the logic* (FO+HS+PIFP) *is* PTIME*-computable.*

Proof. (Sketch.) Fix a relational vocabulary Υ. As in the proof of Lemma 5.1.6, we view formulas as terms and provide for every (FO+HS+PIFP) term t over Υ^+ with $\mathrm{free}(t) = \{\bar{x}\}$ a polynomial-time bounded Turing machine $M_{\Upsilon,t}$ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ and all interpretations $\bar{a}$ of $\bar{x}$ (chosen from the universe of $\mathcal{A}^+$), $M_{\Upsilon,t}$ on input $(\mathcal{A}, \bar{a})$ outputs $t^{\mathcal{A}^+}[\bar{a}]$. The construction of $M_{\Upsilon,t}$ is again by induction on the construction of t. This time, however, $M_{\Upsilon,t}$ may require polynomial space in order to store some of the sets that occur during a computation. Below, we define a set $\mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a}) \in A \cup \mathrm{HF}(A)$ which contains all those sets that are examined by $M_{\Upsilon,t}$ during the computation on input $(\mathcal{A}, \bar{a})$. The definition of $\mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a})$ is by induction on the construction of $t(\bar{x})$, simultaneously for all interpretations $\bar{a}$ of $\bar{x}$:

- If $t(\bar{x}) = x_i$, then $\mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a}) := \mathrm{TC}(a_i)$.
- If $t(\bar{x}) = \varnothing$ or $t(\bar{x}) = \mathit{atoms}$, then $\mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a}) := \mathrm{TC}(t^{\mathcal{A}^+})$.
- If $t(\bar{x}) = f(t_1(\bar{x}), \ldots, t_k(\bar{x}))$, then

$$\mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a}) \;:=\; \bigcup_{i=1}^{k} \mathit{Active}_{t_i(\bar{x})}(\mathcal{A}, \bar{a}).$$

- If $t(\bar{x}) = \{t_1(\bar{x}), \ldots, t_k(\bar{x})\}$, then

$$\mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a}) \;:=\; \{t^{\mathcal{A}^+}[\bar{a}]\} \cup \bigcup_{i=1}^{k} \mathit{Active}_{t_i(\bar{x})}(\mathcal{A}, \bar{a}).$$

- If $t(\bar{x}) = \{t_0(\bar{x}, y) : y \in r(\bar{x}) : \varphi(\bar{x}, y)\}$, then

$$\begin{aligned}\mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a}) \;:=\; & \{t^{\mathcal{A}^+}[\bar{a}]\} \cup \mathit{Active}_{r(\bar{x})}(\mathcal{A}, \bar{a}) \cup \\ & \bigcup_{b \in r^{\mathcal{A}^+}[\bar{a}]} \big[\mathit{Active}_{t_0(\bar{x},y)}(\mathcal{A}, \bar{a}, b) \cup \mathit{Active}_{\varphi(\bar{x},y)}(\mathcal{A}, \bar{a}, b)\big].\end{aligned}$$

- Suppose that $t(\bar{x}) = [\mathrm{IFP}_y\, t_0(\bar{x}, y)]_p$. As in the definition of the semantics of polynomially-bounded IFP terms, define a sequence $(Y_i)_{i\in\omega}$ of sets by induction on i: $Y_0 := \varnothing$ and $Y_{i+1} := Y_i \cup t_0^{\mathcal{A}^+}[\bar{a}, Y_i]$. Let n be the cardinality of A. If there exists a $k \in \omega$ with $Y_k = Y_{k+1} \in \mathrm{HF}_{p(n)}(A)$, then set $m = k$. Otherwise, there exists a unique k with $Y_0, \ldots, Y_k \in \mathrm{HF}_{p(n)}(A)$ and $Y_{k+1} \notin \mathrm{HF}_{p(n)}(A)$; set $m = k + 1$.

$$\mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a}) \;:=\; \{\varnothing\} \cup \bigcup\nolimits_{i=0}^{m} \mathit{Active}_{t_0(\bar{x},y)}(\mathcal{A}, \bar{a}, Y_i).$$

Verify that $\mathrm{TC}(t^{\mathcal{A}^+}[\bar{a}]) \subseteq \mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a})$ and that $\mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a})$ is transitive.

Claim. For every (FO+HS+PIFP) term $t(\bar{x})$ over Υ^+ with $\mathrm{free}(t) = \{\bar{x}\}$ there exists a polynomial $p_t(n)$ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ with universe A, every $k \in \omega$, and all interpretations $\bar{a}$ of $\bar{x}$ chosen from $A \cup \mathrm{HF}_k(A)$

$$|\mathit{Active}_{t(\bar{x})}(\mathcal{A}, \bar{a})| \;\leq\; \max\{p_t(|A|), k\}.$$

The proof of the claim is by an easy induction on the construction of t. Notice that in the case where t is a (FO+HS+PIFP) sentence the claim provides a polynomial $p_t(n)$ such that $|\mathit{Active}_t(\mathcal{A})| \leq p_t(|A|)$ for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$. The details of the construction of the Turing machine $M_{\Upsilon,t}$ are left to the reader. □

Theorem 6.2.4. *Every (three-valued) problem in* $\tilde{\mathrm{C}}$PTIME *is expressible in the logic* (FO+HS+PIFP).

Proof. Consider a problem (C_1, C_2) in $\tilde{\mathrm{C}}$PTIME with $C_1, C_2 \subseteq \mathrm{Fin}(\Upsilon)$ and suppose that the PTime bounded ASM $M = (\Pi, p(n), q(n))$ separates C_1 and C_2, i.e., M accepts every structure in C_1 and rejects every structure in C_2. We are going to define three (FO+HS+PIFP) sentences $\varphi_{M,acc}$, $\varphi_{M,rej}$, and $\varphi_{M,ind}$ over Υ^+ expressing that M accepts, rejects, or is indecisive. More precisely, M accepts (resp. rejects, neither accepts nor rejects) an input $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ iff $\mathcal{A}^+$ is a model of $\varphi_{M,acc}$ (resp. $\varphi_{M,rej}$, $\varphi_{M,ind}$).

Recall from [BGS99] that every PTime bounded ASM is equipped with a boolean output symbol *halt* indicating controlled termination of a computation. W.l.o.g., we can assume that the program of M has the following form:

```
program Π:

  do-for-all x_1 ∈ r_1
     do-for-all x_2 ∈ r_2(x_1)
        ⋱
           do-for-all x_m ∈ r_m(x_1, ..., x_{m-1})
              Π'

  if halt = false then
     ct := ct ∪ {ct}
```

where Π' does not contain `do-for-all`, and ct is a nullary dynamic function symbol not occurring in Π'. Intuitively, ct contains the current time during a run of M. (M is *time-explicit* in the sense of [BGS99].) Fix an enumeration $f_1, \dots, f_d$ of the dynamic symbols of M. A state $\mathcal{S}$ of M is a structure of the form $(\mathcal{A}^+, f_1^{\mathcal{S}}, \dots, f_d^{\mathcal{S}})$ where $\mathcal{A}^+$ is the HF-extension of the current input of M, and $f_1^{\mathcal{S}}, \dots, f_d^{\mathcal{S}}$ are the interpretations of the dynamic symbols of M in state $\mathcal{S}$.

Claim. For every $f \in \{f_1, \dots, f_d\}$ there exists a (FO+HS) formula $update_{\Pi,f}(\bar{x}, y)$ over $\Upsilon^+ \cup \{f_1, \dots, f_d\}$ such that for every state $\mathcal{S}$ of M and all interpretations $\bar{a}, b$ of $\bar{x}, y$ (chosen from the universe of $\mathcal{S}$),

$$\mathcal{S} \models update_{\Pi,f}[\bar{a}, b] \;\Leftrightarrow\; (f, \bar{a}, b) \in \mathrm{Den}'(\Pi, \mathcal{S}) \text{ and } \mathrm{Den}'(\Pi, \mathcal{S}) \text{ is consistent,}$$

where $\mathrm{Den}'(\Pi, \mathcal{S})$ and the corresponding notion of consistency is defined as in [BGS99, Section 4.6].

Proof of the claim. (Sketch.) We call a (FO+HS) term t *critical in* Π if there is an update $(g(t_1, \dots, t_k) := t_0)$ in Π such that $t = t_i$ for some $i \in \{0, 1, \dots, k\}$. Let $t_1, \dots, t_l$ be an enumeration of all terms critical in Π and let

$$critical_\Pi \;:=\; \bigcup \big\{ \{t_i : x_1 \in r_1, \dots, x_m \in r_m\} : 1 \le i \le l \big\}.$$

Verify that this term is closed and definable in (FO+HS). Now follow the definition of the FO formula $update_{\Pi,f}$ in the proof of [BGS99, Lemma 16] but replace every unbounded FO quantifier with the corresponding $critical_\Pi$-bounded quantifier, e.g., replace $\exists x$ with $(\exists x \in critical_\Pi)$ and $\forall x$ with $(\forall x \in critical_\Pi)$. One obtains a (FO+HS) formula as desired. $\Diamond$

Later in the proof it will be important to assume that each $update_{\Pi,f}$ is *flat* in the sense that every occurrence of a dynamic function symbol f_j in $update_{\Pi,f}$ is in the context of an equation of the form $(f_j(\bar{t}) = x)$ where x is a variable. The next claim provides for each $update_{\Pi,f}$ a flat version.

Consider the run $\rho = (\mathcal{S}_i)_{i \le l}$ of M on some input $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$. Let A be the universe of $\mathcal{A}$. We call $a \in A \cup \mathrm{HF}(A)$ *critical in* $\mathcal{S}_i$ if there are $a_0, a_1, \dots, a_k \in A \cup \mathrm{HF}(A)$ such that for some dynamic function symbol f_j, $f_j^{\mathcal{S}_i}(a_1, \dots, a_k) = a_0$, $a_0 \neq \varnothing$, and $a \in \{a_0, a_1, \dots, a_k\}$. We call $a \in A \cup \mathrm{HF}(A)$ *active in* $\mathcal{S}_i$ if $a \in A \cup \{\varnothing, \{\varnothing\}\}$ or if there is an element b critical in $\mathcal{S}_i$ with $a \in \mathrm{TC}(b)$. For every $i \le l$, we denote by $Critical_i$ (resp. $Active_i$) the set of elements critical (resp. active) in at least one of the states $\mathcal{S}_0, \dots, \mathcal{S}_i$. The following facts about $Critical_i$ and $Active_i$ will be important later on. Let n be the cardinality of A. By definition of runs of M, $l \le p(n)$ and $|Active_l| \le q(n)$. It is easy to see that for every $i \le l$, $Active_i$ is transitive and, if we let $Critical'_i := Critical_i \cup A \cup \{\varnothing, \{\varnothing\}\}$, then

$$Active_i \;=\; \mathrm{TC}(Critical'_i) - \{Critical'_i\} \tag{6.1}$$

$$|Active_i| \;=\; |\mathrm{TC}(Critical'_i)| - 1. \tag{6.2}$$

Claim. Let $update_{\Pi,f}(\bar{x}, y)$ be defined according to the first claim. There exists a flat (FO+HS) formula $update^*_{\Pi,f}(\bar{x}, y, z)$ over $\Upsilon^+ \cup \{f_1, \ldots, f_d\}$ such that for every run ρ as above, every $i \leq l$, and all interpretations $\bar{a}, b$ of $\bar{x}, y$ (chosen from the universe of $\mathcal{S}_i$)

$$\mathcal{S}_i \models update_{\Pi,f}[\bar{a}, b] \leftrightarrow update^*_{\Pi,f}[\bar{a}, b, Critical_i].$$

The crux in the proof of this claim is to replace subterms of the form $f_j(\bar{t})$ that occur in a 'non-flat' context with a new variable x that ranges in $Critical_i$ and to add a 'flat' guard of the form $(f_j(\bar{t}) = x)$. For instance, if $\{f_j(\bar{t}) : y \in r : \varphi\}$ is a subterm of $update_{\Pi,f}$, then replace this subterm with the term $\{x : y \in r, x \in z : \varphi \wedge f_j(\bar{t}) = x\}$ and let the variable z be interpreted as $Critical_i$. We omit the details of the proof. Note that similarly one can obtain from the closed (FO+HS) term $critical_\Pi$ in the proof of the first claim a flat version $critical^*_\Pi(z)$ satisfying $critical^{*\,\mathcal{S}_i}_\Pi[Critical_i] = critical_\Pi{}^{\mathcal{S}_i}$ for every state $\mathcal{S}_i$ in ρ.

The following inductive definition of $(Critical_i)_{i \leq l}$ is inspired by the proof of [BGS99, Theorem 18]:

$$\begin{aligned} C_0 &:= \varnothing \\ C_{i+1} &:= C_i \cup \big\{(i+1, j, \bar{a}, b) : \\ &\qquad j \in \{1, \ldots, d\},\ \bar{a}, b \in Critical_i \cup critical^{\mathcal{S}_i}_\Pi : \\ &\qquad \chi(C_i, i, j, \bar{a}, b)\big\}, \end{aligned}$$

where

$$\begin{aligned} \chi(C_i, i, j, \bar{a}, b) &:= \big(\mathcal{S}_i \models update_{\Pi,f_j}[\bar{a}, b] \wedge b \neq \varnothing\big) \vee \\ &\qquad \big((i, j, \bar{a}, b) \in C_i \wedge \neg(\exists b' \in critical^{\mathcal{S}_i}_\Pi)\mathcal{S}_i \models update_{\Pi,f_j}[\bar{a}, b']\big). \end{aligned}$$

We can view each C_i as a subset of $\mathrm{HF}(A)$ if we view k-tuples over $A \cup \mathrm{HF}(A)$ as nested ordered pairs. (Ordered pairs can be defined according to the standard Kuratowski definition; see, e.g., [BGS99, Section 6].) Let $proj^l_k$ be a (FO+HS) term denoting a function that maps every k-tuple to the l-th component of that tuple. With the help of $proj^l_k$ one can define a (FO+HS) term $extract(z)$ such that $extract^{\mathcal{A}^+}[C_i] = Critical_i$ for every $i \leq l$. Hence, $Critical_i$ in the induction step of the above definition of $(C_i)_{i \in \omega}$ can be replaced with $extract^{\mathcal{A}^+}[C_i]$. Next, we remove all explicit occurrences of $\mathcal{S}_i$ in the definition of $(C_i)_{i \in \omega}$.

Let $currentTime(z)$ be a (FO+HS) term such that $currentTime^{\mathcal{A}^+}[C_i] = i$ for every $i \leq l$. For every $f \in \{f_1, \ldots, f_d\}$, let $update^*_{\Pi,f}(\bar{x}, y, z)$ be obtained according to the second claim. Obtain $update^{**}_{\Pi,f}(\bar{x}, y, z)$ from $update^*_{\Pi,f}(\bar{x}, y, z)$ as follows:

1. replace every occurrence of z with $extract(z)$, and

2. replace every occurrence of an equation $(f_j(\bar{t}) = x)$ with the (FO+HS) formula

$$(ct, j, \bar{t}, x) \in z \vee \big[x = \varnothing \wedge \neg\big(\exists x' \in \mathit{extract}(z)\big)(ct, j, \bar{t}, x') \in z\big],$$

where ct abbreviates $\mathit{currentTime}(z)$.

Now verify that for every state $\mathcal{S}_i$ in ρ

$$\mathcal{S}_i \models \mathit{update}_{\Pi,f}[\bar{a}, b] \;\Leftrightarrow\; \mathcal{A}^+ \models \mathit{update}^{**}_{\Pi,f}[\bar{a}, b, C_i].$$

In the same manner one obtains from the flat version $\mathit{critical}^*_\Pi(z)$ of $\mathit{critical}_\Pi$ a (FO+HS) term $\mathit{critical}^{**}_\Pi(z)$ satisfying $\mathit{critical}^{**}_\Pi{}^{\mathcal{A}^+}[C_i] = \mathit{critical}_\Pi{}^{\mathcal{S}_i}$ for every state $\mathcal{S}_i$ in ρ. Here is a refurbished inductive definition of $(C_i)_{i\in\omega}$:

$$\begin{aligned} C_0 &:= \varnothing \\ C_{i+1} &:= C_i \cup \mathit{nextStage}^{\mathcal{A}^+}[C_i], \end{aligned}$$

where the (FO+HS) term $\mathit{nextStage}(z)$ over Υ^+ is defined as follows:

$$\begin{aligned} \mathit{nextStage}(z) &:= \big\{(i+1, j, \bar{x}, y) : \\ &\quad j \in \{1, \ldots, d\},\ i, \bar{x}, y \in \mathit{extract}(z) \cup \mathit{critical}^{**}_\Pi(z) : \\ &\quad \underbrace{i = \mathit{currentTime}(z) \wedge \chi'(z, i, j, \bar{x}, y)}_{\gamma}\big\} \\ i+1 &:= i \cup \{i\} \\ \chi'(z, i, j, \bar{x}, y) &:= \big(\mathit{update}^{**}_{\Pi,f_j}(\bar{x}, y, z) \wedge y \neq \varnothing\big) \vee \\ &\quad \big((i, j, \bar{x}, y) \in z \wedge \neg(\exists y' \in \mathit{critical}^{**}_\Pi(z))\mathit{update}^{**}_{\Pi,f_j}(\bar{x}, y', z)\big). \end{aligned}$$

This shows that each $\mathit{Critical}_i$ is (FO+HS)-definable, e.g., by $\mathit{extract}[z/c_i]$ where c_i is a (FO+HS) term defined inductively as follows: $c_0 := \varnothing$ and $c_{i+1} := c_i \cup \mathit{nextStage}[c/c_i]$.

Next, we redefine the guard γ in the definition of *nextStage* so that (the semantics of) c_i becomes stable if the time bound $p(n)$ or the bound $q(n)$ on the number of activated elements is exceeded. To this end, define $\mathit{Critical}'(z)$ to be the (FO+HS) term

$$\mathit{extract}[z/z \cup \mathit{nextStage}(z)] \cup \mathit{atoms} \cup \{\varnothing, \{\varnothing\}\}.$$

Recall equality (6.2) and verify that for every $i < l$

$$\begin{aligned} \big[\mathrm{IFP}_x\, \mathit{Critical}'[z/c_i]\big]_{q+1}{}^{\mathcal{A}^+} \neq \varnothing &\;\Leftrightarrow\; \mathit{Critical}'_{i+1} \in \mathrm{HF}_{q(n)+1}(A) \\ &\;\Leftrightarrow\; |\mathrm{TC}(\mathit{Critical}'_{i+1})| \leq q(n)+1 \\ &\;\Leftrightarrow\; |\mathit{Active}_{i+1}| \leq q(n). \end{aligned}$$

Let *nextStage*$'(z)$ be defined as *nextStage*(z), except that now the guard γ is replaced with

$$\gamma \wedge [\mathrm{IFP}_x\ \mathit{Critical}'(z)]_{q+1} \neq \varnothing \wedge [\mathrm{IFP}_x\ i+1]_{p+1} \neq \varnothing.$$

One can now choose a polynomial $r(n)$, independent of $\mathcal{A}$, such that $[\mathrm{IFP}_z\ \mathit{nextStage}'(z)]_r$ defines C_l. In other words, if we set $c^* = [\mathrm{IFP}_z\ \mathit{nextStage}'(z)]_r$, then for every input $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, $c^{*\mathcal{A}^+}$ is an encoding of the run of M on $\mathcal{A}$, and $\mathit{extract}[z/c^*]^{\mathcal{A}^+}$ is the set of critical elements in that run. We come to the definition of the (FO+HS+PIFP) sentences $\varphi_{M,acc}$, $\varphi_{M,rej}$, and $\varphi_{M,ind}$ expressing that M accepts, rejects, or is indecisive:

$$\begin{aligned}\varphi_{M,acc} \;:=\; (\exists z, z' \in c^*)\big(&\mathit{proj}^1(z) = \mathit{proj}^1(z') = \mathit{currentTime}(c^*) \wedge \\ &\mathit{proj}^2(z) = \text{“halt”} \wedge \mathit{proj}^3(z) = \mathit{true} \wedge \\ &\mathit{proj}^2(z') = \text{“output”} \wedge \mathit{proj}^3(z') = \mathit{true}\big).\end{aligned}$$

$\varphi_{M,rej}$ and $\varphi_{M,ind}$ are defined similarly. □

A Negative Result

One of the main results in [BGS99] is that the following PTIME-computable boolean query is not computable by any PTime bounded ASM and is thus not in $\tilde{\mathrm{C}}$PTIME. Let S be a set symbol.

SUBSET-PARITY : Given some $\mathcal{A} \in \mathrm{Fin}(\{S\})$, decide whether $|S^{\mathcal{A}}|$ is even.

Using the results in [BGS99] we can show the following theorem.

Theorem 6.2.5. SUBSET-PARITY *is not definable in* (FO+HS+PIFP).

In order to prove the theorem we need to introduce some additional notation. Fix a distinguished constant symbol *inputSize*. Let $\mathcal{A}$ be a finite structure over some relational vocabulary Υ, and let n be the cardinality of the universe of $\mathcal{A}$. By $(\mathcal{A}^+, |A|)$ we denote the structure over $\Upsilon^+ \cup \{\mathit{inputSize}\}$ obtained from $\mathcal{A}^+$ by adding as interpretation of *inputSize* the von Neumann ordinal for n. Notice that the constant $\mathit{inputSize}^{(\mathcal{A}^+,|A|)}$ is a set in $\mathrm{HF}(\varnothing)$. By $\mathrm{L}^{\omega}_{\infty,\omega}$ we denote *finite variable infinitary logic* and by $\mathrm{L}^{m}_{\infty,\omega}$ the m-variable fragment of $\mathrm{L}^{\omega}_{\infty,\omega}$ (see, e.g., [EF95]).

Lemma 6.2.6. *Let Υ be a relational vocabulary. For every* (FO+HS+PIFP) *sentence φ over $\Upsilon^+ \cup \{\mathit{inputSize}\}$ there exists an $\mathrm{L}^{\omega}_{\infty,\omega}$ sentence φ' over $\Upsilon \cup \{\in, \varnothing, \mathit{inputSize}\}$ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, $(\mathcal{A}^+, |A|) \models \varphi \leftrightarrow \varphi'$.*

Aside from the well-known fact that fixed-point iterations are definable in $\mathrm{L}^{\omega}_{\infty,\omega}$ (see, e.g., [BGS99, EF95]), the proof of Lemma 6.2.6 is based on the following

observation. Let $p(n)$ be a polynomial. There exists an $\mathrm{L}^{\omega}_{\infty,\omega}$ formula $\mathit{size}_p(x)$ over $\{\in, \varnothing, \mathit{inputSize}\}$ such that for every finite structure $\mathcal{A}$ with universe A, and for every set $X \in \mathrm{HF}(A)$

$$(\mathcal{A}^+, |A|) \models \mathit{size}_p[X] \quad \Leftrightarrow \quad X \in \mathrm{HF}_{p(|A|)}(A).$$

The proof of Lemma 6.2.6 is rather technical and omitted here.

Proof of Theorem 6.2.5. (Sketch.) It suffices to show that there is no (FO+HS+ PIFP) sentence φ over $\{S\}^+ \cup \{\mathit{inputSize}\}$ such that for every $\mathcal{A} \in \mathrm{Fin}(\{S\})$, $(\mathcal{A}^+, |A|) \models \varphi$ iff $\mathcal{A}$ is a positive instance of SUBSET-PARITY. We point out that this is a consequence of the proof of [BGS99, Corollary 45] and Lemma 6.2.6. Recall the definition of $\mathit{Active}_\varphi(\mathcal{A})$ in the proof of Lemma 6.2.3 and set $\mathit{Active}'_\varphi(\mathcal{A}) = \mathit{Active}_\varphi(\mathcal{A}) \cup A$. Now check that [BGS99, Corollary 33] remains true if "PTime program Π" is replaced with "(FO+HS+PIFP) sentence φ", and "$\mathit{Active}_\Pi(\mathcal{I})$" is replaced with "$\mathit{Active}'_\varphi(\mathcal{I})$". The only change occurs in the proof of [BGS99, Theorem 24] where we have to provide an upper bound n^k on the cardinality of $\mathit{Active}'_\varphi(\mathcal{I})$ (for sufficiently large $\mathcal{I}$). Such a bound exists by the claim in the proof of Lemma 6.2.3. We can now follow the proof of [BGS99, Corollary 45], using Lemma 6.2.6 instead of [BGS99, Theorem 18], and derive a contradiction from the assumption that there exists a sentence φ defining SUBSET-PARITY in the above sense. □

Corollary 6.2.7. (FO+HS+PIFP) *does not capture* PTIME.

7

Discussion

In this second part of the thesis, we have studied applications of ASMs to the theory of computation. Motivated by the observation that the standard notion of logarithmic-space reducibility on structures, which is based on Turing machines manipulating sting encodings of structures, suffers from a fundamental drawback, namely the lack of an 'appropriate' encoding, we have put forward an encoding-free notion of logarithmic-space computability based on ASMs. This notion can serve as a basis for a reduction theory among structures and has naturally led to the definition of a new complexity class which can be regarded as the logarithmic-space counterpart of Choiceless Polynomial Time ($\tilde{\text{C}}$PTIME) [BGS99]. We have shown that this new class, called Choiceless Logarithmic Space ($\tilde{\text{C}}$LOGSPACE), is a proper subclass of both LOGSPACE and $\tilde{\text{C}}$PTIME, thereby separating LOGSPACE and PTIME on the choiceless level. Taking the results of [GM95] and [BGS99] into account, we have obtained the following relations between standard, choiceless, and descriptive complexity classes:

LOGSPACE	$\subseteq$	PTIME	(standard)
$\cup^{(1)}$		$\cup^{(1)}$	
$\tilde{\text{C}}$LOGSPACE	$\subset^{(2)}$	$\tilde{\text{C}}$PTIME	(choiceless)
$\cup^{(3)}$		$\cup^{(2)}$	
(FO+DTC)	$\subset^{(3)}$	(FO+LFP)	(descriptive)

where the inclusions marked (1), (2), and (3) are proper due to the following decision problems:

(1) SUBSET-PARITY (see Corollary 6.1.5 and [BGS99, Section 10]).

(2) SMALL-SUBSET-PARITY (see Corollary 6.1.6 and [BGS99, Section 7]).

(3) Reachability in double graphs (see Lemma 5.2.5 and [GM95]).

There is certain mismatch between $\tilde{\text{C}}$LOGSPACE and $\tilde{\text{C}}$PTIME because $\tilde{\text{C}}$LOGSPACE is an ordinary (i.e., two-valued) complexity class, while $\tilde{\text{C}}$PTIME is a three-valued class (recall the discussion at the beginning of Section 3.4). A possible solution to this problem was suggested by Blass, Gurevich, and Shelah in [BGS99]. They proposed to equip PTime bounded ASMs with a counting function and claimed that the extended model can detect when a polynomial-time bound expires. This would one allow to define $\tilde{\text{C}}$PTIME in a natural way as a two-valued class. Indeed, the fact that all $\tilde{\text{C}}$PTIME problems are expressible in the logic (FO+HS+PIFP) indicates that the machines of the extended model will be able to trigger their own termination once a polynomial-time bound expires (recall the proof of Theorem 6.2.4 and notice that (FO+HS+PIFP) also has a kind of counting function built-in, namely one that determines the size of a given set). Thus, a natural way to continue this investigation would be to redefine $\tilde{\text{C}}$PTIME as a two-valued class and to check whether the obtained class is captured by the logic (FO+HS+PIFP).

Bibliography

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

[AHVdB96] S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal Versus First-Order Logic to Query Temporal Databases. In *Proceedings of 15th ACM Symposium on Principles of Database Systems (PODS '96)*, pages 49–57. ACM Press, 1996.

[AV89] S. Abiteboul and V. Vianu. Fixpoint Extensions of First-Order Logic and Datalog-Like Languages. In *Proceedings of 4th IEEE Symposium on Logic in Computer Science (LICS '89)*, pages 71–79, 1989.

[AV91] S. Abiteboul and V. Vianu. Generic Computation and its Complexity. In *Proceedings of 23th ACM Symposium on Theory of Computing (STOC '91)*, pages 209–219. ACM Press, 1991.

[AVFY98] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational Transducers for Electronic Commerce. In *Proceedings of 17th ACM Symposium on Principles of Database Systems (PODS '98)*, pages 179–187. ACM Press, 1998.

[AY96] N.R. Adam and Y. Yesha. Electronic Commerce: An Overview. In N.R. Adam and Y. Yesha, editors, *Electronic Commerce*, volume 1028 of *Lecture Notes in Computer Science*, pages 5–12. Springer-Verlag, 1996.

[BG87] A. Blass and Y. Gurevich. Existential Fixed-Point Logic. In E. Börger, editor, *Computation Theory and Logic*, volume 270 of *Lecture Notes in Computer Science*, pages 20–36. Springer-Verlag, 1987.

[BGG97] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.

[BGR95] E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In E. Börger, editor,

Specification and Validation Methods, pages 231–243. Oxford University Press, 1995.

[BGS99] A. Blass, Y. Gurevich, and S. Shelah. Choiceless Polynomial Time. Technical Report MSR-TR-99-08, Microsoft Research, 1999.

[BGVdB99] A. Blass, Y. Gurevich, and J. Van den Bussche. Abstract State Machines and Computationally Complete Query Languages. Technical Report MSR-TR-99-95, Microsoft Research, 1999.

[BH98] E. Börger and J. Huggins. Abstract State Machines 1988–1998: Commented ASM Bibliography. *Bulletin of the EATCS*, 64:105–127, February 1998. See also `http://www.eecs.umich.edu/gasm`.

[Bör99] E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullman, editors, *Current Trends in Applied Formal Methods (FM-Trends '98)*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer Verlag, 1999.

[Büc60] J.R. Büchi. Weak Second-Order Arithmetic and Finite Automata. *Z. Math. Logik Grundlagen Math.*, 6:66–92, 1960.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic. *ACM Trans. on Prog. Lang. and Sys.*, 8(2):244–263, April 1986.

[CH82] A. Chandra and D. Harel. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[Com93] K. J. Compton. A Deductive System for Existential Least Fixpoint Logic. *Journal of Logic and Computation*, 3(2):197–213, 1993.

[DEGV97] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. In *Proceedings of 12th Annual IEEE Conference on Computational Complexity (CCC '97)*, pages 82–101, 1997.

[EF95] H. D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, 1995.

[EFT94] H. D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer-Verlag, 2nd edition, 1994.

[Eme90] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–11072. Elsevier Science Publishers B.V., 1990.

[FAY97] B. Fordham, S. Abiteboul, and Y. Yesha. Evolving Databases: An Application to Electronic Commerce. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, August 1997.

[GHR95] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation – P-Completeness Theory.* Oxford University Press, 1995.

[GKS91] G. Gottlob, G. Kappel, and M. Schrefl. Semantics of Object-Oriented Data Models – The Evolving Algebra Approach. In J. W. Schmidt and A. A. Stogny, editors, *Next Generation Information Technology*, volume 504 of *Lecture Notes in Computer Science*, pages 144–160. Springer-Verlag, 1991.

[GM95] E. Grädel and G. McColm. On the Power of Deterministic Transitive Closures. *Information and Computation*, 119:129–135, 1995.

[GR00] Y. Gurevich and D. Rosenzweig. Partially Ordered Runs: A Case Study. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Proceedings of International Workshop on Abstract State Machines (ASM 2000)*, volume 1912 of *Lecture Notes in Computer Science*, pages 131–150. Springer-Verlag, 2000.

[Grä92] E. Grädel. On Transitive Closure Logic. In *Proceedings of 5th Workshop on Computer Science Logic (CSL '91)*, volume 626 of *Lecture Notes in Computer Science*, pages 149–163. Springer-Verlag, 1992.

[Gro94] M. Grohe. *The Structure of Fixed-Point Logics.* PhD thesis, Universität Freiburg, 1994.

[Gur88] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[Gur91] Y. Gurevich. Evolving Algebras: An Attempt to Discover Semantics. *Bulletin of the EATCS*, 43:264–284, 1991. See also: G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Gur97a] Y. Gurevich. From Invariants to Canonization. *Bulletin of the EATCS*, 63:115–115, 1997.

[Gur97b] Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan, May 1997.

[Gur99] Y. Gurevich. The Sequential ASM Thesis. *Bulletin of the EATCS*, 67:93–124, 1999.

[HK84] D. Harel and D. Kozen. A Programming Language for the Inductive Sets, and Applications. *Information and Control*, 63:118–139, 1984.

[Hod93] W. Hodges. *Model Theory*. Cambridge University Press, 1993.

[Imm86] N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Control*, 68:86–104, 1986.

[Imm87] N. Immerman. Languages That Capture Complexity Classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.

[Imm98] N. Immerman. *Descriptive Complexity*. Springer Graduate Texts in Computer Science, 1998.

[IV97] N. Immerman and M.Y. Vardi. Model Checking and Transitive Closure Logic. In *Proceedings of 9th International Conference on Computer-Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 291–302. Springer-Verlag, 1997.

[LMSS93] A. Levy, I. Mumick, Y. Sagiv, and O. Shmueli. Equivalence, Query-Reachability, and Satisfiability in Datalog Extensions. In *Proceedings of 12th ACM Symposium on Principles of Database Systems (PODS '93)*, pages 109–122, 1993.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[PV98] P. Picouet and V. Vianu. Semantics and Expressiveness Issues in Active Databases. *Journal of Computer and System Sciences*, 57(3):325–355, 1998.

[Ros99] E. Rosen. An Existential Fragment of Second Order Logic. *Archive for Mathematical Logic*, 38:217–234, 1999.

[SC85] A.P. Sistla and E.M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the Association for Computing Machinery*, 32(3):733–749, July 1985.

[Var82] M. Vardi. The Complexity of Relational Query Languages. In *Proceedings of 14th ACM Symposium on Theory of Computing (STOC '82)*, pages 137–146. ACM Press, 1982.

[Var95] M. Vardi. On the Complexity of Bounded-Variable Queries. In *Proceedings of 14th ACM Symposium on Principles of Database Systems (PODS '95)*, pages 266–276. ACM Press, 1995.

Index