Complexity Theory via Abstract State Machines

von
Antje Nowack
Matrikelnummer 205 162

Diplomarbeit

in Informatik

vorgelegt der
Fakultät für Mathematik, Informatik und Naturwissenschaften der
Rheinisch-Westfälischen Technischen Hochschule Aachen
im Juni 2000

Angefertigt am
Lehr- und Forschungsgebiet für
Mathematische Grundlagen der Informatik
bei
Professor Dr.phil. Erich Grädel

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Structure Of This Work . . . 2
1.3 Acknowledgements . . . 3

2 The ASM-Model **5**
2.1 Vocabularies, States, Variable Assignments, Updates . . . 5
2.2 Deterministic Rules, Programs, Runs . . . 7
2.3 Non-Deterministic Rules, Programs, Runs . . . 10
2.4 Connection Between The Two Kinds Of Non-Determinism . . . 13
2.5 Normal Forms . . . 14
2.6 Remarks . . . 19

3 Computational Complexity In General **21**
3.1 Complexity Classes . . . 21
3.2 Diagonalization Theorem . . . 23
3.3 Some Useful Definitions . . . 27

4 Time Complexity **29**
4.1 Deterministic Time Measures . . . 30
4.2 Non-Deterministic Time Measures . . . 31
4.3 Macrostep-Complexity-Classes . . . 33
4.4 Microstep-Complexity-Classes . . . 39
4.5 Consequences Of The Diagonalization Theorem . . . 41
4.6 Determinism And Non-Determinism . . . 44
4.7 The Classes P And NP . . . 48
4.8 A Logic Captured By P . . . 51
4.9 A Logical Description Of NP_{el} . . . 52
4.10 Completeness . . . 59

5 Space Complexity **71**
5.1 Deterministic Space . . . 71
5.2 Non-Deterministic Space . . . 74

5.3 Determinism And Non-Determinism 75
5.4 Relations Between Complexity Classes 75
5.5 Completeness 79

6 Alternating Abstract State Machines 83
6.1 The Basic Model 83
6.2 Alternating Complexity Classes 85

7 Probabilistic Abstract State Machines 91

8 Simulating Other Computation Models by ASMs 97
8.1 Turing Machines 98
8.2 Random Access Machines 100
8.2.1 The Basic Model 100
8.2.2 Simulating RAMs 102
8.3 BSS-Machines 111
8.3.1 The Basic Model 112
8.3.2 Simulating BSS-machines 113
8.3.3 Remark 118
8.4 Consequences 118

9 Conclusions 123

Chapter 1

Introduction

1.1 Motivation

The standard computation model is Turing machines. The inputs of a Turing machine are strings over a fixed alphabet. But the problems, to which algorithms are applied, mostly do not deal with strings but with structures, e.g. graphs. So, a structure must be encoded as a string over a fixed alphabet to give it to a Turing machine. This encoding makes it more difficult to understand a Turing machine. An advantage of this model is that it is rather simple and theoretically easy to handle.
Originally, complexity theory has been introduced via Turing machines (e.g., see [12], [14]). The notion "classical", used in the following, will refer to this.
Abstract State Machines (ASMs) is another computation model. They have been introduced first in [8] where they are called "evolving algebras" (later they have been renamed to "Abstract State Machines"). Other introducing works are [9] and [10] ASMs directly work on the structures, so that an encoding of the structure is not necessary anymore. The actions are updates of the structures or to be more precise sets of updates. They are described by a rule resp. program. These actions are more complex than the transitions of a Turing machine. Most problems can be modeled on their natural abstraction level. Furthermore, it is much much easier to program such a machine than to define a transition function of a Turing machine which does the intended. Therefore, this model is much more spread in praxis than Turing machines.
"Complexity Theory Via ASMs" means that complexity is not defined via Turing machines but via ASMs. I.e., complexity measures for ASMs (time and space) are defined and similar considerations to those in classical complexity theory will be done. The question is whether complexity theory can be introduced via ASMs instead of Turing machines in spite of the more complex construction or whether complexity theory becomes too difficult to handle.
[1] and [3] deal with special aspects of complexity theory via ASMs. In [1], the

Linear Time Hierarchy Theorem (for a certain time measure which will be defined later) has been proven. It does not hold for Turing machines with the usual time measure. This indicates that complexity theory via ASMs is not the same as via Turing machines and that one has consider it separately. This work gives an introduction to this kind of complexity theory and finally considers the question for the connection to complexity theory via other computation models.

1.2 Structure Of This Work

An ASM-model similar to the one of [1], extended by a background part and two kinds of non-determinism, will be used in this work. The first kind of non-determinism gives the possibilty to guess an element from the universe of the structure, the second kind gives the possibility to guess a rule to be executed. This model will be introduced in chapter 2. It is the basis of this work.
In all chapters, the general case and a special case of (input) structures will be considered. In this special case, no static functions besides the logic ones are allowed. This leads to structures with a finite living part which makes them easier to handle.
In chapter 3, computational complexity in general will be introduced. The resources for measuring complexity will not be fixed in this chapter. It will be described how an ASM can be encoded by a structure of a certain vocabulary. Such a structure can be the input for an ASM or part of an input. With this, a theorem with a proposition similar to the classical diagonalization theorem will be proven.
In chapter 4, time complexity will be considered. Two measures for time will be defined: via macrosteps and via microsteps. So, there will be six kinds of complexity classes: these two complexity measures respectively combined with determinism, rule-non-determinism, element-non-determinism. Starting from this, some results will be proven which are partly similar and partly different from those in classical theory. It will turn out that it is a big difference which kind of non-determinisim is considered. The two time measures lead partly to different results and partly it does not matter which time measure has been chosen. E.g., if polynomial time is considered then it does not matter what time measure has been chosen because if the needed time is polynomial for one measure then it is also polynomial for the other measure. Time measured in macrosteps and time measured in microsteps differ from each other only by a constant factor for a fixed ASM. But indeed, there are differences between these two measures. For macrosteps, the Speed-Up-Theorem (analogous formulation to the classical case) holds but for microsteps it is not. The invalidity of the Speed-Up-Theorem for microsteps is a consequence of the Linear Hierarchy Theorem proven in [1]. Furthermore, some consequences of the diagonalization theorem will be proven, namely hierarchy results. The connection between determinism and non-determinism

(both kinds) will be considered. This leads to the complexity classes P, NP_{rule} and NP_{el} (they will be defined analogously to the classes P and NP in the classical case). Afterwards, a theorem with a proposition similar to Fagin's theorem in the classical will be proven for element-non-determinism. I.e., a logic similar to existential second order logic will be defined and afterwards it will be proven that the set of all sentences this logic corresponds to NP_{el} on structures with an ordered active part. At the end of this chapter, some completeness results will be proven. Namely, for P, NP_{rule}, NP_{el}.
In chapter 5, space complexity will be onsidered but this does not lead to many results in the general case. The reason is that even if a bound for the space is given (e.g., by a function), one can not guarantee that the run does not contain infintely many non-isomorphic states. An example will witness this.
In chapter 6, alternating ASMs will be defined. They will be derived from "ordinary" non-deterministic ASMs in a way which is similar to the derivation of alternating Turing machines from "ordinary" non-deterministic Turing machines. This definition is possible for both kinds of non-determinism, and the two possibilities lead to different results again. Most results deal with the connection between deterministic complexity and alternating complexity.
In chapter 7, probabilistic ASMs will be defined. In the case of element-non-determinism, probabilistic ASMs will be not derived from "ordinary" element-non-determinstic ASMs as simply as probabilistic Turing machines are derived from "ordinary" non-deterministic Turing machines. With every structure, one has to connect a probability measure on the potential set of the living part of the universe of the structure. An important point is to preserve stochastic independence. Another problem is that there might be non-empty sets with probability zero. Results similar to those in the classical case will be proven with some restrictions (mostly) by the existence of non-empty sets with probability zero.
In chapter 8, it will be proven that Turing machines, Random Access Machines and BSS-machines (a generalized version) can be strictly lock-step-simulated by ASMs. Furthermore the connection between complexity theory via ASMs and complexity theory via the other computation models will be considered.
Chapter 9 contains the conclusions.

1.3 Acknowledgements

My special thank is directed to Professor Dr. Erich Grädel. He has suggested this interesting subject and accompanied the development of this work attentively.
Furthermore, I thank Diplom-Informatiker Marc Spielmann for the critical correcture and the useful remarks and Diplom-Informatiker Stephan Kreutzer for the review.
Finally, I want to thank Professor Dr. Wolfgang Thomas for the willingness to execute the co-referate.

Chapter 2

The ASM-Model

In this chapter, the ASM-model, which is the basis of this work, is described. First, some basic definitions are recalled. Most of this chapter is similar to [1] and [9].

2.1 Vocabularies, States, Variable Assignments, Updates

Vocabulary. A *vocabulary* is a finite set of relation and function names of fixed arities. Relations are treated as special functions as follows. Every element of the vocabulary is a function name but some of them are tagged as *relational* . Some of the function names are tagged as *static.* Furthermore it is assumed that every vocabulary contains the nullary functions Mode and Output and the following logic function names: nullary names true, false and undef, a unary relation name Boole, the names of the standard Boolean operations (i.e., $\vee, \wedge, \neg$) and the equality sign. Let voc be the set of all these vocabularies.
In the following, for a vocabulary $\Upsilon \in \text{voc}$, let Υ_{stat} be the set of all function names in Υ tagged as static, and let $\Upsilon_{\text{dyn}} := \Upsilon - \Upsilon_{\text{stat}}$ be the set of all non-static (and therefore dynamic) function names in Υ.

Term. *Terms* are defined as in first-order logic. *Boolean-valued terms* are Boolean combinations of terms $P(\bar{x})$ where P is a relation name (i.e., a function name tagged relational). A *ground* term is a term without free variables.

State. A *state* A of vocabulary Υ is an Υ-structure (i.e., A consists of a set X, which is called the superuniverse of A and the interpretations of the function names in Υ on X, which are called the basic functions of A). An r-ary function is interpreted as a function from X^r to X. The nullary function names true, false, and undef are interpreted by different elements. Concerning these three elements, there is no difference between the names and their interpretations in

the following.
The only values for a function tagged relational are true and false. The equality relation is interpreted as usual (as the identity relation). In the following, a relation (a function tagged relational) R is identified with the set $\{\bar{x} : R(\bar{x}) = \text{true}\}$. In particular, Boole $= \{\text{true}, \text{false}\}$.
Via the nullary function undef, it is possible to deal with partial functions. A basic function f in state A is undefined at a tuple $\bar{a}$ if $f^A(\bar{a}) = \text{undef}$. The Boolean connectives are undefined if at least one of the arguments is not Boolean; over Boole they are defined as usual.

Reserve. The superuniverse includes an infinite and countable set called reserve of A. It can be defined as the greatest subset of X satisfying the following conditions:

- No function produces a reserve value.
- Every basic relation, with the exception of the equality relation, evaluates to false if at least one of the arguments is in the reserve.
- Every other function evaluates to undef if at least one of the arguments is in the reserve.

The elements of the universe which are not in the reserve are called *living*.

Expanded State. The variables considered here are individual variables ranging over the universe of a given state A. Let V be a collection of variable names. A *variable assignment* with domain V over the state A is an injetive map $\xi : V \to \text{universe}(A)$. The pair (A, ξ) is called an *expanded state*.

Expansion. Let A be a state of vocabulary Υ and $f_1, ..., f_n \notin \Upsilon$ be function names with arities $r_1, ..., r_n$ respectively. Then the expansion of A by $f_1, ..., f_n$ ($\text{Exp}(A, f_1, ..., f_n)$) is a state of vocabulary $\Upsilon \cup \{f_1, ..., f_n\}$ where the universe is the same as the one of A, the elements of Υ are interpreted as in A and $f_i(x_1, ..., x_{r_i}) = \text{undef}$ ($i \in \{1, ..., n\}$) for all $x_1, ..., x_{r_i}$ from the universe of A (so f_i is defined nowhere).
Let L_1, L_2 be sets of Υ-structures. Then $\text{Exp}(L_1, f_1, ..., f_n) := \{\text{Exp}(A, f_1, ..., f_n) : A \in L_1\}$ and $\text{Exp}((L_1, L_2), f_1, ..., f_n) := (\text{Exp}(L_1, f_1, ..., f_n), \text{Exp}(L_2, f_1, ..., f_n))$.

Active domain. Let I be an Υ-structure. The active domain of an r-ary function $f \in \Upsilon$ in I is the set of all element $x \neq \text{undef}$ in the universe ofm I such that at least one of the following two conditions holds:

1. There exists an r-tuple $\bar{y}$ of elements of the universe of I such that $f^I(\bar{y}) = x$.

2. There exists an r-tuple $\bar{y}$ of elements of the universe of I such that x is equal to at least one of the components of $\bar{y}$ and $f^I(\bar{y}) \neq$ undef.

2.2 Deterministic Rules, Programs, Runs

The rough idea of the computation model "ASM" is the following: The input is a structure over a given vocabulary. During the computation of an ASM on a given input, the structure is changed, i.e., the interpretations of the function names. The output is the last structure of the computation or, in the case of a Boolean query, the interpretation of Output in the last state. The updates of the structure are described by a rule resp. a program which is applied repeatedly until it not possible anymore.
Syntax. The set of ASM-rules is defined inductively as follows:

- An *atomic* (or update) rule R is an expression $f(t_1, ..., t_r) := t_0$ where each t_i is a term and f is a non-static r-ary function name; if f is a relation name then t_0 is a Boolean-valued term.

- A *block* rule has the form

  `block`
  R_1
  .
  .
  .
  R_k
  `endblock`

  where each R_i is a rule.

- A *conditional* rule R has the form

  `if` g `then` R' `endif`

  where g is a Boolean-valued term and R' is a rule. A line "`if` g `then` R' `endif`" is a clause with the guard g and the body R.

- An *import* rule R has the form

  `import` v
  R_0
  `endimport`

  where v is a variable and R_0 is a rule and it is the body of R.

Free and bound variables and occurences of variables are defined as usual with "`import` v" binding v.
After having defined the syntax, the semantics of ASM-rules will be defined in the following. For this, some preparational thoughts have to be done.

Content. A state A can be considered as a map from locations to contents. *Locations* have the form $(f, \bar{a})$ where f is a function name of some arity r and $\bar{a}$ is an r-tuple of elements of the superuniverse. The *content* of $(f, \bar{a})$ is $f^A(\bar{a})$.

Update. An *update* of a state A is a pair $\alpha = (l, a)$ where l is a location over A and a an element of A. Executing this update means to put a into l (i.e., set the context of l to a) and leave the other locations as they are. This will be called "to *fire* (l, a).

Contradictory. A set U of updates of a state A is *contradictory* if it contains a pair of updates (l, a) and (l, b) where $a \neq b$; otherwise U is *consistent*. To fire a contradictory update set, do nothing; the new state is identical to the old one. To execute a consistent update set $\{(l_1, a_1), ..., (l_k, a_k)\}$, put elements $a_1, ..., a_k$ into locations $l_1, ..., l_k$ respectively and leave the other locations as they are.

Semantics. The evaluation of a (deterministic) *rule* R at a state A under a variable assignment ξ produces a (deterministic) update set $U = \mathrm{US}(R, A, \xi)$. To fire R, fire U; the result is the successor of A with respect to R. R is consistent (respectively contradictory) at (A, ξ) if U is so.
Now, consider the update set produced by a rule:

- Given an atomic rule as described above and an expanded state (A, ξ), evaluate all terms t_i. Let a_i be the resulting value of t_i, $\bar{a} = (a_1, ..., a_r)$ and l the location $(f, \bar{a})$. Then $\mathrm{US}(R, A, \xi) = \{(l, a_0)\}$.

- Given a block rule as described above and an expanded state (A, ξ), evaluate all rules R_i. $\mathrm{US}(R, A, \xi) = \bigcup_{1 \leq i \leq k} \mathrm{US}(R_i, A, \xi)$.

- Given a conditional rule as described above and an expanded state (A, ξ), evaluate the guard. If the guard evaluates to false, then $\mathrm{US}(R, A, \xi) = \emptyset$. If g evaluates to true, then $\mathrm{US}(R, A, \xi) = \mathrm{US}(R', A, \xi)$.

- Given an import rule as described above and a state A and a variable assignment ξ with domain V, let $V' = V \cup \{v\}$ (where v is an element of the reserve) and $\xi[v \to a]$ be the variable assignment with domain V' obtained from ξ by picking a fresh (that is outside of the range of ξ) reserve element a and setting or resetting $\xi[v \to a](v) = a$. Evaluate R_0 at $(A, \xi[v \to a])$. $\mathrm{US}(R, A, \xi) = \mathrm{US}(R_0, A, \xi[v \to a])$.

Two elements imported at the same time must be inequal. E.g., to every import rule, an infinite subset of the reserve is assigned from which the element is taken when the respective rule is executed. If the rules are different then the corresponding sets are disjoint.

Program. A *program* is a rule without free variables. So, it can be fired at a state, with the empty assignment, and this firing produces a state. Thus it makes sense to fire a program repeatedly, producing a sequence of states.

Vocabulary. The *vocabulary of a rule* is the set of all function names appearing in the rule.
A *rule* r *over a vocabulary* Υ is a rule such all elements of Υ appear in r and the vocabulary of r is Υ.

State. A state of a program P is any state of the vocabulary of P. Let $\mathrm{States}(P)$ be the set of all states of P.
Connected to every program P, there is a subset of $\mathrm{States}(P)$ containing all states of P which can be an input for P. These structures are called input structures of P. Let $\mathrm{Inputstates}(P) \subseteq \mathrm{States}(P)$ be the set of all states I of P such that $\mathrm{Mode}^I = \mathrm{true}$ and $\mathrm{Output}^I = \mathrm{false}$.

Active part and background part For a given structure I (over vocabulary Υ), one can define two subsets of $\mathrm{Living}(I)$: $\mathrm{Active}(I)$ and $\mathrm{Background}(I)$.
$\mathrm{Background}(I)$ is union of the active domains of all static functions in I.
$\mathrm{Active}(I)$ is the union of the active domains of all dynamic functions in I.
The active structure of I is defined as the structure induced by the active part and restricted to the dynamic functions.
The backgound structure of I is defined as the structure induced by the background part and restricted to the static functions.
Note that for a given structure I, $\mathrm{Background}(I)$ and $\mathrm{Active}(I)$ are not necessarily disjoint.
Furthermore, $\mathrm{Boole} = \{\mathrm{true}, \mathrm{false}\}$ is always a subset of the background part (the only functions defined in Boole are $\wedge$, $\vee$, $\neg$).

ASM. An ASM is a pair (P, Input) where P is an ASM-program and $\mathrm{Input} \subseteq \mathrm{Inputstates}(P)$. Input is the set of all input structures for $\mathcal{A}$. All states in Input must have the same background structure.
$\mathcal{A} = (P, \mathrm{Input})$ is an ASM over Υ if P is a program over Υ and Input does only contain Υ-structures.
According to the definition of an "input structure for an ASM", the input contains both parts, the active part and the backgound part. Because, the background structure does not change during the computation, the active structure is the real input.

Final, enabling, disabling. A state A is *final* if Mode $=$ false at A. In the case of a Boolean query, the output of a final state is the value of Output at A. Else, the state A itself is the output.
Let P be a program. The states of P split into *enabling* states where P is enabled and *disabling* states where P is disabled. A state A is enabling if it is non-final and the update set U of P at A (with the empty assignment) is consistent; otherwise A is disabling. Any final state is disabling. Non-final disabling states are *hang* states.

Successor. For a deterministic program P, $A \vdash_P A'$ is true if $\mathrm{US}(P, A, \xi)$ transforms the state A into the state A'. The state A' is the successor of A with respect to P. Furthermore, let $\vdash_P^k$ be the k-fold iteration of $\vdash_P$, let $\vdash_P^+$ be the transitive closure of $\vdash_P$, and let $\vdash_P^*$ be the reflexive transitive closure of $\vdash_P$. A state A of P is *reachable* if $I \vdash_P^* A$ for some input state I.

Run. Let ρ be a sequence $(A_i)_{i<\lambda}$ of states of P where λ ranges over the natural numbers and the first infinite ordinal ω, which is the collection of all natural numbers (so, for every ordinal $\alpha < \omega$ iff $\alpha \in \omega$).

- ρ is a *partial run* of P if $A_i \vdash_P A_{i+1}$ for all i with $i + 1 < \lambda$. If λ is finite, then the number $\lambda - 1$ is the length of ρ.
- A partial run ρ is a *run* if A_0 is an input state and if λ is finite, then $A_{\lambda-1}$ is disabling.
- Suppose that ρ is a run. If λ is finite and the state $A_{\lambda-1}$ is final, then ρ *is converging* otherwise it is *diverging*. If ρ is finite but P diverges on A_0, then ρ is *hanging*.
- A deterministic ASM-program P converges on A_0 if the run of P on A_0 is converging, otherwise it diverges on A_0. If P converges on A_0, then the output yielded by $A_{\lambda-1}$ is the output of P on A_0. If the run of A_0 is hanging, then *hangs* on A_0.

Accept. P *accepts* (respectively *rejects*) an input state I if it converges on I and $\mathrm{Output}^I = \mathrm{true}$ (respectively $\mathrm{Output}^I = \mathrm{false}$).
If $\mathcal{A} = (P, \mathrm{Input})$ is a deterministic ASM over Υ, then $\mathcal{A}$ defines the following set of Υ-structures:
$\mathrm{Acc}(\mathcal{A}) := \{I : I \in \mathrm{Input} \text{ and } I \text{ is accepted by } P\}$

2.3 Non-Deterministic Rules, Programs, Runs

In this section, non-deterministic ASMs are defined. Again, they are defined via rules.

Syntax. The definition of non-determinstic rules for ASMs is obtained by adding the following two points to the definition in the deterministic case, i.e., the syntax for atomic rules, block rules, conditional rules and import rules is defined as in the deterministic case:

- A *guess-element* rule has the form

  `guess-element` v : $g(v)$
  $R_0(v)$
  `endguess-element`

  where v is a variable, $g(v)$ a Boolean-valued term and $R_0(v)$ a non-deterministic rule. v is the head variable, $g(v)$ is the guard (a Boolean-valued term) and R_0 is the body.
  In [9], this rule has been introduced as choose rule.

- A *guess-rule* rule has the form

  `guess-rule`
  R_1
  R_2
  `endguess-rule`

  where R_1 and R_2 are non-determinstic rules.
  In [9], this rule has been introduced as choose-among rule or explicit choice.

Semantics. The evaluation of a non-deterministic *rule* R at a state A under a variable assignment ξ produces a non-deterministic update set $U = \mathrm{NUS}(R, A, \xi)$ or to be more precise, a set of deterministic update sets. To fire R, fire one update set from U; the result is a successor of A with respect to R. R is consistent at (A, ξ) if one of the elements of U is so. R is contradictory at (A, ξ) if all elements of U are so.
Now, consider the non-deterministic update-set produced by a rule:

- Given an atomic rule as described above and an expanded state (A, ξ), evaluate all terms t_i. Let a_i be the resulting value of t_i, $\bar{a} = (a_1, ..., a_r)$ and l the location $(f, \bar{a})$. Then $\mathrm{NUS}(R, A, \xi) = \{\{(l, a_0)\}\}$.

- Given a block rule as described above and an expanded state (A, ξ) then $\mathrm{NUS}(R, A, \xi) = \{\bigcup_{1 \leq i \leq k} X_i : X_i \in \mathrm{NUS}(R_i, A, \xi)\}$.

- Given a conditional rule as described above and an expanded state (A, ξ), evaluate the guard. If the guard evaluates to false, then $\mathrm{NUS}(R, A, \xi) = \{\emptyset\}$. And if g evaluates to true, then $\mathrm{NUS}(R, A, \xi) = \mathrm{NUS}(R', A, \xi)$.

- Given an import rule as described above, a state A and a variable assignment ξ with domain V, let $V' = V \cup \{v\}$ (where v is an element of the reserve) and $\xi[v \to a]$ be the variable assignment with domain V' obtained from ξ by picking a fresh (that is outside of the range of ξ) reserve element a and setting or resetting $\xi[v \to a](v) = a$. Evaluate R_0 at $(A, \xi[v \to a])$. $\mathrm{NUS}(R, A, \xi) = \mathrm{NUS}(R_0, A, \xi[v \to a])$.
  As in the deterministic case, two elements imported at the same time must be inequal. This is achieved as before.

- Given a guess-element rule as described above and an expanded state (A, ξ). Let $M := \{a \in \mathrm{universe}(A) : a \text{ is non-reserve and } A \models g[v/a]\}$.
  $\mathrm{NUS}(R, A, \xi) = \bigcup_{a \in M} \mathrm{NUS}(R_0, A, \xi[v \to a])$ where $\xi[v \to a]$ for an $a \in M$ is obtained as $\xi[v \to a]$ from ξ in the case of the import rule. This kind of non-determinism is called *element-non-determinism*.

- Given a guess-element rule as described above and an expanded state (A, ξ). Then $\mathrm{NUS}(R, A, \xi) = \mathrm{NUS}(R_1, A, \xi) \cup \mathrm{NUS}(R_2, A, \xi)$.
  This kind of non-determinism is called *element-non-determinism*.

Free and bound variables and occurences of variables are defined as usual with "`import` v" and "`guess-element` v : $g(v)$" binding v.
A *non-deterministic program* is a non-deterministic rule without free variables.

States. States, States(P) for a non-deterministic program P, Inputstates(P) for a non-deterministic program P are defined as in the deterministic case.

Active part and background part The active part, the background part and the respective structures are defined as in the deterministic case.

Non-deterministic ASM. A non-deterministic ASM is a pair (P, Input) where P is a non-deterministic ASM-program and $\mathrm{Input} \subseteq \mathrm{States}(P)$. Input is the set of all input structures for $\mathcal{A}$.
$\mathcal{A} = (P, \mathrm{Input})$ is a non-deterministic ASM over Υ if P is a non-determinstic program over Υ and Input does only contain Υ-structures.

Properties of states. *Final*, *enabling*, *disabling* and *hang* states are defined a in the deterministic case.

Successor. For a non-deterministic program P $A \vdash_P A'$ is true if there is a set $X \in \mathrm{NUS}(P, A, \xi)$ of updates that transforms the state A into the state A'. The state A' is a successor of A with respect to P.

Run. $\vdash_P^k$, $\vdash_P^+$, $\vdash_P^*$, the notions *reachable*, *partial run*, *run* and their properties *converging*, *diverging* and *hanging* are derived from $\vdash_P$ as in the deterministic case.

Accept. If P is a non-deterministic program, then P *accepts* an input state I if there is a converging run of P on I and the output yielded by the last state of the run is true ($\text{Output}^I = \text{true}$).
If $\mathcal{A} = (P, \text{Input})$ is a non-deterministic ASM over Υ, then $\mathcal{A}$ defines the following set of Υ-structures:
$\text{Acc}(\mathcal{A}) := \{I : I \in \text{Input and } I \text{ is accepted by } P\}$

Successor with respect to guesses. Furthermore, for a non-deterministic program P and a state A of P, one can define the following: Let m be the number of guess-element rules and n the number of guess-rule rules in P. Then A' is the successor of I with respect to P, a_1, ..., a_m, b_1, ..., b_n, where a_1, ..., a_m are elements of the universe of A and $b_1, ..., b_n \in \{0, 1\}$ iff the updates determined by the guesses a_1, ..., a_m, b_1, ..., b_n lead from A to A'.

2.4 Connection Between The Two Kinds Of Non-Determinism

In this section, the connection between the two kinds of non-determinism given by the guess-element rule and the guess-rule rule is considered.

Theorem 2.1. *Rule-non-determinism can be simulated by element-non-determinism.*

Proof. Given a rule of the following form:

```
guess-rule
   R1
   R2
endguess-rule
```

This rule-non-deterministic rule can be expressed by help of the guess-element rule as follows (this has already been mentioned in [9]):

```
guess-element v :  Boole(v)
   if v then R1
   if ¬v then R2
endguess-element
```

□

So, the guess-element rule is at least as powerful as the guess-rule rule (and the overhead concerning time is not too big as one will see later. No additional space is needed).

Theorem 2.2. *Element-non-determinism can not be simulated by element-non-determinism.*

Proof. Consider $\Upsilon = \{\wedge, \vee, \text{true}, \text{false}, \neg, \text{undef}\} \cup \{0, 1, +, \cdot, <\} \cup \{f\}$ such that $\Upsilon_{\text{stat}} = \{\wedge, \vee, \text{true}, \text{false}, \neg, \text{undef}\} \cup \{0, 1, +, \cdot, <\}$, $\Upsilon_{\text{dyn}} = \{f\}$ and $\wedge$, $\vee$, true, false, $\neg$, undef are as usual, $0, 1$ are nullary function names, $+$, $\cdot$, $<$ are binary function names, f is a unary function name and $<$ is tagged as relational.
Furthermore, consider the following pair (L_1, L_2) of Υ-structures of finite size: $L_2 = \{I : I$ is an Υ-structure of finite size, the background strucutre of I is $(\mathbb{R}, 0, 1, +, \cdot, <)\}$.
$L_1 = \{I \in L_2 : f^I(\sqrt{2}) \neq \text{undef}\}$
where $\sqrt{2}$ is the real number x such that $x \cdot x = 2$ and $0 < x$.
For the element-non-deterministic ASM $\mathcal{A} = (P, L_2)$ where

```
guess-element v :   (v · v = 1 + 1) ∧ 0 < v
  block
    if ¬(f(v) = undef) then Output := true endif
    if f(v) = undef then Output := false endif
    Mode := false
  endblock
endguess-element
```

it is true that $\text{Acc}(\mathcal{A}) = L_1$.
But $\sqrt{2}$ is not accessible and therefore, no such rule-non-deterministic ASM does exist. ☐

2.5 Normal Forms

In this section, for every kind of ASM-program (deterministic, element-non-determinism, rule-non-determinism) one normal form will be described. Every ASM-program is equivalent to an ASM-program to an ASM-program in the respective normal form.

Definition 2.3. Two programs P and P' are equivalent if the the following holds:
For every finite run $s_0 \vdash_P s_1 \vdash_P ... \vdash_P s_n$ of P on s_0 there is a run of P' on $s_0 = s'_0 \vdash_{P'} s'_1 \vdash_{P'} ... \vdash_{P'} s'_n$ of the same length such that s_i and s'_i are isomorphic for $i \in \{0, ..., n\}$.
For every infinite run $s_0 \vdash_P s_1 \vdash_P ...$ of P on s_0 there is an infinite run of P' on $s_0 = s'_0 \vdash_{P'} s'_1 \vdash_{P'} ...$ such that s_i and s'_i are isomorphic for $i \in \mathbb{N}$.
and the other direction does also hold.
For two rules, equivalence is defined in the same way but instead of states, expanded states are used.

Definition 2.4. Two ASMs $\mathcal{A} = (P, L)$ and $\mathcal{A}' = (P', L')$ are equivalent if P and P' are equivalent and $L = L'$.

W.l.o.g., one can assume that an ASM program does not reuse variables, i.e., each occurence of `import` v or `guess-element` $v:\quad g(v)$ has a different variable v.

Lemma 2.5. *Every determinstic ASM-program is equivalent to a deterministic ASM-program of the following form:*

```
import v_1
  .
  .
  .
import v_k
  block
    if c_1 then R_1 endif
          .
          .
          .
    if c_m then R_m endif
  endblock
endimport
  .
  .
  .
endimport
```

where R_i*,* $i \in \{1, ..., m\}$ *are updates.*

Proof. This has already been proven in [6] (where it is called "second normal form"). The proof in [6] is the same as the proof of lemma 2.6 but without the parts for the element-non-deterministic rule. □

Lemma 2.6. *Every element-non-deterministic ASM-program is equivalent to a non-determinstic ASM-program of the following form:*

```
import v_1
  .
  .
  .
import v_l
  guess-element w_1 :  g_1(w_1)
    .
    .
    .
  guess-element w_m :  g_m(w_m)
    block
      if c_1 then R_1 endif
        .
        .
        .
      if c_n then R_n endif
    endblock
  endguess-element
    .
    .
    .
  endguess-element
endimport
  .
  .
  .
endimport
```

where R_i, $i \in \{1, ..., m\}$ are updates.

Proof. The proof will be done by transforming an arbitrary rule-non-determinstic ASM-program into an ASM-program of the described form.
Because of the assumption, that variables are not reused, two variable used in the same rule are distinct variables and because two import rules import different elements at the same time.

original rule	transformation

if g then
import v
R
endimport
endif
	import v
	if g then
	R
	endif
	endimport

import v R endimport import w S endimport	import v import w R S endimport endimport
guess-element v : $g(v)$ R endguess-element guess-element w : $h(w)$ S endguess-element	guess-element v : $g(v)$ guess-element w : $h(w)$ R S endguess-element endguess-element
guess-element v : $g(v)$ import w R endimport endguess-element	import w guess-element v : $g(v)$ R endguess-element endimport
import v R endimport guess-element w : $g(w)$ S endguess-element	import v guess-element w : $g(w)$ R S endguess-element endimport
guess-element v : $g(v)$ R endguess-element import w S endimport	import w guess-element v : $g(v)$ R S endguess-element endimport

if g then
if h then R endif
endif
	if $g \wedge h$ then R endif

original rule	transformation

`if c then`
  `block`
    R_1
    .
    .
    .
    R_n
  `endblock`
`endif`
	`block`
	  `if` c `then` R_1 `endif`
	  .
	  .
	  .
	  `if` c `then` R_n `endif`
	`endblock`

`if` c `then`
  `guess-element` v `:` $g(v)$
    R
  `endguess-element`
`endif`
	`guess-element` v`:` $c \to g(v)$
	  `if` c `then` R `endif`
	`endguess-element`

The following comment does refer to those transformations where two rules R and S occur. Because variables are not reused, v does not occur in S and w does not occur in R. Therefore, also in these cases, no collisions appear.
Applying these transformations to a given program until it is not possible anymore, leads to an equivalent ASM-program of the described form. □

Lemma 2.7. *Every rule-non-deterministic ASM-program is equivalent to a non-determinstic ASM-program of the following form:*

```
import v_1
  .
  .
  .
import v_k
  guess-element w_1 :  Boole(w_1)
    .
    .
    .
  guess-element w_l :  Boole(w_l)
    block
      if c_1 then R_1 endif
        .
        .
        .
      if c_m then R_m endif
    endblock
  endguess-element
    .
    .
    .
  endguess-element
endimport
  .
  .
  .
endimport
```

where R_i, $i \in \{1, ..., m\}$ are updates.

Proof. The proof can be done similarly to the one of lemma 2.6 with only a few differences. □

2.6 Remarks

Reserve. In the following, complexity aspects will be considered (after having defined the respective notions). Often, it is necessary to calculate the number of reachable states within a limited time or space (time and space will be defined in the later chapters). This has to be read as "up to isomorphism" because executing an import rule leads to infinitely many successors but they are identical apart from the imported elements. Taking the imported elements into account, they are isomorph because they are taken from the reserve which is a naked set. But the difference between these two states does not really interest.

Formally, one could find a solution for this problem by identifying the reserve with the natural numbers with order (given by a successor function) and zero. Executing an import rule means taking the next element which has not been imported until now.

Computation graph. For later purposes, define the notion of a computation graph:
Let $\mathcal{A} = (P, \text{Input})$ be a non-deterministic ASM and $I \in \text{Input}$. The *computation graph* $\mathcal{G}_{\mathcal{A},I}$ is the directed graph $G = (V, E)$ such that:

1. V is the set of all states reachable from I by use of P (where two states are identified if the only difference is caused by importing different elements from the reserve).

2. B is a successor of A iff $A \vdash_P B$.

In the deterministic case, the graph is a path.

Chapter 3

Computational Complexity In General

3.1 Complexity Classes

To deal with complexity, one has to measure the size of an input. Because, starting from here, one will only deal with structures I such that Active(I) is finite. The intuition of the following definition of size is the following: The real input does only consist of the active part of the universe of the structure and the interpretations of the dynamic funtions. Every element of the active part does occur in the active domain of the interpretation of a dynamic function name. Thus, it makes sense to define the size of the structure as the sum of the sizes of the interpretations of the dynamic function names where the size of an interpretation of a dynamic function is the size of its domain assumed that it is finite. The domain of the interpretation of a function name f ($\mathrm{dom}(f^I)$) is the set of all $\mathrm{arity}(f)$-tuples for which f is not interpreted as undef.

Definition 3.1. Let $\Upsilon \in \mathrm{voc}$, I be an Υ-structure such that Active(I) is finite. The *size* size(I) of I is given by $\mathrm{size}(I) := \sum_{f \in \Upsilon_{\mathrm{dyn}} - \{\mathrm{Mode}, \mathrm{Output}\}} |\mathrm{dom}(f^I)|$ where $\mathrm{dom}(f^I)$ is the subset of $(\mathrm{Active}(I))^{\mathrm{arity}(f)}$ where f^I is defined (i.e., is not equal to undef).

In the rest of this work, often the term *structure of finite size*. This means the following: A structure I is of finite size if Active(I) is finite and therefore, size(I) is well-defined.

Note that the background of a structure of finite size might be infinite.

This size measure can also be called "unit cost measure" because all tuples of elements and elements of the universe (except undef and the elements of the reserve) are weighted equally.

The nullary function names Mode and Output are not weighted because every vocabulary, considered in this context, does contain them. For input structures they even have always the same values.

Lemma 3.2. *Let* $\Upsilon \in \mathrm{voc}$ *and* I *be an* Υ*-structure of finite size. Then the following inequation holds:*

$$|\mathrm{Active}(I)| \leq |\Upsilon_{\mathrm{dyn}}| \cdot \max_{g \in \Upsilon_{\mathrm{dyn}}} (\mathrm{arity}(g) + 1) \cdot \mathrm{size}(I)$$

Proof. Let $\Upsilon \in \mathrm{voc}$ and I be an Υ-structure of finite size. Consider $g \in \Upsilon_{\mathrm{dyn}}$ of arity r. Then every r-tuple, where g is not interpreted as undef, can increase $|\mathrm{Active}(I)|$ by at most $r + 1$. Every element of $\mathrm{Active}(I)$ has to appear in the domain or the range of the interpretation of a dynamic function name.
Therefore, the following holds:

$$\begin{aligned} |\mathrm{Active}(I)| &\leq \sum_{g \in \Upsilon_{\mathrm{dyn}}} (\mathrm{arity}(g) + 1) \cdot \mathrm{size}(I) \\ &\leq |\Upsilon_{\mathrm{dyn}}| \cdot \max_{g \in \Upsilon_{\mathrm{dyn}}} (\mathrm{arity}(g) + 1) \cdot \mathrm{size}(I) \end{aligned}$$

□

Lemma 3.3. *Let* $\Upsilon \in \mathrm{voc}$ *and* I *be an* Υ*-structure of finite size. Then the following inequation holds:*

$$\mathrm{size}(I) \leq |\Upsilon_{\mathrm{dyn}} - \{\mathrm{Mode}, \mathrm{Output}\}| \cdot |\mathrm{Active}(I)|^{\max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g))}$$

Proof. Let $\Upsilon \in \mathrm{voc}$ and I be an Υ-structure of finite size. Consider $g \in \Upsilon_{\mathrm{dyn}}$ of arity r. There are $\leq |\mathrm{Active}(I)|^r$ r-tuples (of elements from $\mathrm{Active}(I)$) for which g is not interpreted as undef. So, the size of the interpretation of g in I is $\leq |\mathrm{Active}(I)|^r$.
Therefore, the following holds:

$$\begin{aligned} \mathrm{size}(I) &\leq \sum_{g \in \Upsilon_{\mathrm{dyn}} - \mathrm{Mode}, \mathrm{Output}} |\mathrm{Active}(I)|^{\mathrm{arity}(g)} \\ &\leq |\Upsilon_{\mathrm{dyn}} - \{\mathrm{Mode}, \mathrm{Output}\}| \cdot |\mathrm{Active}(I)|^{\max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g))} \end{aligned}$$

□

The following definition is a formalization of the notion "complexity class". For better understanding, the intuitive meaning of the components of the definition will be explained. R is a resource which is used during the computation, e.g. time or space or something else. How much is used, depends on the input and the program and one has to measure for the use. T is a mapping limiting the use of the resource depending on the size of the input.

Definition 3.4. Let $\Upsilon \in$ voc, $T : \mathbb{N} \to \mathbb{N}$ and R be a mapping that maps pairs consisting a pair consisting of a program and structure to a natural number. A *complexity class* for $\Upsilon \in$ voc is defined as follows:
$R_\Upsilon(T) := \{(L_1, L_2) : L_2$ is a set of Υ-structures of finite size such that $\mathrm{Mode}^I =$ true and $\mathrm{Output}^I =$ false for all $I \in L_2$, $L_1 \subseteq L_2$, there exists an ASM $\mathcal{A} = (P, L_2)$ such that $\mathrm{Acc}(\mathcal{A}) = L_1$ and $R(P, I) \leq T(\mathrm{size}(I))$ for all $I \in L_2\}$
Starting from this, one can define the following:
$R(T) := \bigcup_{\Upsilon \in \mathrm{voc}} R_\Upsilon(T)$

Definition 3.5. Let $\Upsilon, \Upsilon' \in$ voc, $\Upsilon' - \Upsilon = \{f_1, ..., f_n\}$, (L_1, L_2) be a pair of sets of Υ-structures of finite size, (L_1', L_2') be a pair of sets of Υ-structures of finite size. (L_1', L_2') is r-reducible to (L_1, L_2) if there is exists an ASM $\mathcal{A} = (A, \mathrm{Exp}(L_1, f_1, ..., f_n)$ satisfying the property r which computes for every $I \in L_2$ a structure $I' \in L_2'$ such that $I \in L_1$ iff $I' \in L_1'$.

Definition 3.6. Let C be a (complexity) class. Let (L_1, L_2) be a pair of a sets of structures. (L_1, L_2) is C-hard via r-reduction if every $(L_1', L_2') \in C$ can be reduced to (L_1, L_2) via r-reduction.

Definition 3.7. Let C be a (complexity) class. Let (L_1, L_2) be a pair of sets of structures of finite size. (L_1, L_2) is *C-complete via r-reduction* if $(L_1, L_2) \in C$ and (L_1, L_2) is C-hard via r-reduction.

3.2 Diagonalization Theorem

For some purposes, it is necessary to encode programs such that the encoding of a program can be an input for another program. For ASMs the inputs are structures. Most of the following definitions are extensions of definitions in [1] They show, how an ASM-program can be coded by a structure which then is an input state for another program.

Definition 3.8. A state A is *initial* if it is isomorphic to a state A_0 such that the active structure of A_0 satisfies the following conditions.

- The active elements without true, false, undef form a segment $[0, n)$ of the natural numbers with $n \geq 2$, the name 0 has the obvious interpretation and Succ is the successor function. Furthermore, $\mathrm{Succ}(n-1) = \mathrm{Succ}(\mathrm{undef}) =$ undef.

- $\mathrm{Mode}^{A_0} =$ true and $\mathrm{Output}^{A_0} =$ false

A_0 is the canonic version of A.
A *parse tree* is a labeled tree with a total labeling function Type and a partial labeling function Nomen.

Definition 3.9. Let e be a syntactical entity. The *abstract parse tree* for e APT(e) is defined inductively as follows:

- If e is a variable, then APT(e) is a one-node tree. The type of (that is the value of the function Type at) the root is " Var ". The nomen of (that is the value of the function Nomen at) the root is "e" itself.
- If e is a term $f(t_1, ..., t_r)$, then APT(e) is the composition of the parse trees for $t_1, ..., t_r$. The type of the root is "Fun" and the nomen is "f".
- If e is an atomic rule $f(t_1, ..., t_r) := t_{r+1}$, then APT($e$) is the composition of the parse trees for $t_1, ..., t_r, t_{r+1}$. The type of the root is "Update"and the nomen is "f".
- If e is a block rule with constituent rules $R_1, ..., R_k$, then APT(e) is the composition of the parse trees for $R_1, ..., R_k$. The type of the root is "Block".
- If e is an if-clause with guard g and body R, then APT(e) is the composition of the parse trees for g and R. The node type of the root is "Clause".
- If e is an import rule with variable v, body R and T is the abstract parse tree of R, then APT(e) is obtained as follows. First, construct the composition of T (all by itself; that is add a parent to the root of T). Let x be the new root. Second, set the type of x to "Import" and the nomen of x to x itself. Third, for every node y of T whose nomen is v, redefine the nomen of y to x.
- If e is a guess-element rule with variable v, guard $g(v)$, body R, T_1 is the abstract parse tree for $g(v)$ and T_2 is the abstract parse tree for R, then APT(e) is obtained as follows. First, construct the composition of T_1 and T_2. Let x be the new root. Second, set the type of x to "Guess-Element" and the nomen of x to x itself. Third, for every node y of $T_1$and every node y of T_2, whose nomen is v, redefine the nomen of y to x.
- If e is a guess-rule rule (rule-non-determinism) with the two (constituent) rules R_1 and R_2, then APT(e) is the composition of the parse trees for R_1 and R_2. The type of the root is "Guess-Rule".

Fix a vocabulary and linearly order it arbitrarily. For every function name f in Υ (including the logic names), denote by f-code the ordinal number of f in the linear order. Let P be a program in the vocabulary Υ.

Definition 3.10. The *concrete parse tree* $\mathrm{CPT}_\Upsilon(P)$ of P is the presentation of the abstract parse tree APT(P) of P obtained as follows.

- Order the nodes of APT(P) in the depth-first manner and then replace every node with its ordinal number (starting with zero).

- Modify the type function, by replacing the labels Block, Clause, Fun, Import, Update, Var, Guess-Element, Guess-Rule with numbers 0, 1, 2, 3, 4, 5, 6, 7 respectively.
- Modify the nomen function by replacing every nomen f with its f-code.

Fix $\Upsilon \in$ voc containing (in addition to 0, Succ, Mode and Output) the unary function names Parent, FC ("First Child"), NS ("Next Sibling"), Type and Nomen and let P be a program of vocabulary Υ. Define the size of P to be the number of nodes in the concrete parse tree $\mathrm{CPT}_\Upsilon(P)$ of P. W.l.o.g., attention is restricted to programs whose size is at least 6 and at least as large as the total number of function names in Υ (including the logic names).

Definition 3.11. An initial state A of vocabulary Υ *encodes* an ASM $\mathcal{A} = (P, \mathrm{Input})$ over Υ if the canonic version A_0 of A satisfies the following conditions:

- The active elements with functions Parent, FC, NS, Type, Nomen form $\mathrm{CPT}_\Upsilon(P)$. On the non-active elements, these functions are undefined.
- Any function name $f \in \Upsilon_{\mathrm{dyn}}$ different from 0, Succ, Mode, Output, Parent, FC, NS, Type and Nomen, is interpreted in the default way, that is, every $f(\bar{a}) = \mathrm{false}$ if f is relational and every $f(\bar{a}) = \mathrm{undef}$ otherwise.
- The background structure of A is the background structure which is common to all elements of Input.

Definition 3.12. Let $\Upsilon \in$ voc and $\mathcal{A}$ be an ASM over Υ.
$I(\Upsilon, \mathcal{A})$ is defined as the canonic initial state encoding $\mathcal{A}$.

Let $\mathcal{A} = (P, \mathrm{Input})$. Given $I(\Upsilon, \mathcal{A})$ and $I \in$ Input (in one structure; the background structure is shared by them). Then the work of $\mathcal{A}$ on I can be simulated rather directly. Go through the tree representing P and execute the respecective instructions. In the case that $I = I(\Upsilon, \mathcal{A})$, one has to construct a copy of the active structure of $I(\Upsilon, \mathcal{A})$ first and then execute the simulation. The copy is necessary because in general, the input structure is changed during the computation but the program resp. the ASM is not changed. A detailed description of the simulation in the special case is given in [1]. This gives directly the simulation in the general case.

Definition 3.13. A vocabulary is called *diagonalization-solvent* if, in addition to the obligatory function names (the logic function names and the names 0, Succ, Mode and Output), it contains the nullary function names C, LastUpdate and unary function names Parent, FC, NS, Type, Nomen, Val, NextUpdate. Of course, it may contain arbitrarily many additional names.

Definition 3.14. Let Υ be a diagonalization-solvent vocabulary and L be a set of Υ-structures of finite size such that $\text{Mode}^I = true$, $\text{Output}^I = \text{false}$ for all $I \in L$. Let $R_\Upsilon(T)$ and $r_\Upsilon(t)$ be complexity classes.
$R_\Upsilon(T)$ *allows diagonalization* over $r_\Upsilon(t)$ concerning L iff there exists an ASM $\mathcal{A}_{\text{Diag}} = (\text{Diag}, L)$ such that:

- $R(\text{Diag}, I) \leq T(\text{size}(I))$ for all $I \in L$ and $\mathcal{A}_{\text{Diag}}$ halts on every structure from L.

- For any ASM $\mathcal{A} = (P, \text{Input})$ over Υ such that $\text{Input} \subseteq L$, $I(\Upsilon, \mathcal{A}) \in \text{Input}$ and $r(P, I) \leq t(\text{size}(I))$ for all $I \in \text{Input}$, the following holds:

$$I(\Upsilon, \mathcal{A}) \in \text{Acc}(\mathcal{A}_{\text{Diag}}) \text{ iff } I(\Upsilon, \mathcal{A}) \notin \text{Acc}(\mathcal{A})$$

Theorem 3.15. *Let Υ be a diagonalization-solvent vocabulary and L be a set of Υ-structures of finite size such that $\text{Mode}^I = true$, $\text{Output}^I = false$ for all $I \in L$. If $R_\Upsilon(T)$ allows diagonalization over $r_\Upsilon(t)$ concerning L, then $(R_\Upsilon(T) \cap \{(\tilde{L}, L) : \tilde{L} \subseteq L\}) - (r_\Upsilon(t) \cap \{(\tilde{L}, L) : \tilde{L} \subseteq L\}) \neq \emptyset$.*

Proof. Let L be a set of Υ-structures of finite size such that $\text{Mode}^I = true$, $\text{Output}^I = \text{false}$ for all $I \in L$ and Let $R_\Upsilon(T)$ and $r_\Upsilon(t)$ be complexity classes such that $R_\Upsilon(T)$ allows diagonalization over $r_\Upsilon(t)$ concerning L.
The proof will be done by contradiction. Because $R_\Upsilon(T)$ allows diagonalization over $r_\Upsilon(t)$ concerning L, there is an ASM $\mathcal{A}_{\text{Diag}} = (\text{Diag}, L)$ satisfying the conditions from definition 3.14. Suppose that "$r_\Upsilon(t) \cap \{(\tilde{L}, L) : \tilde{L} \subseteq L\} = R_\Upsilon(T) \cap \{(\tilde{L}, L) : \tilde{L} \subseteq L\}$". So, $(L_1, L) = (\text{Acc}(\mathcal{A}_{\text{Diag}}), L) \in r_\Upsilon(t) \cap \{(\tilde{L}, L) : \tilde{L} \subseteq L\}$ because (as a consequence of the propertis of Diag) $(L_1, L) \in R_\Upsilon(T) \cap \{(\tilde{L}, L) : \tilde{L} \subseteq L\}$. Furthermore, for every program $\mathcal{A} = (P, \text{Input})$ over Υ such that $\text{Input} \subseteq L$, $I(\Upsilon, \mathcal{A}) \in \text{Input}$ and $r(P, I) \leq t(\text{size}(I))$ for all $I \in \text{Input}$ the following holds:

$$\text{Diag accepts } I(\Upsilon, \mathcal{A}) \text{ iff } P \text{ rejects } I(\Upsilon, \mathcal{A})$$

Because $(L_1, L) \in r_\Upsilon(t)$ (and $r_\Upsilon(t) \cap \{(\tilde{L}, L) : \tilde{L} \subseteq L\} = R_\Upsilon(T) \cap \{(\tilde{L}, L) : \tilde{L} \subseteq L\}$), there exists an ASM $\mathcal{A}'_{\text{Diag}} = (\text{Diag}', L)$ such that $\text{Acc}(\mathcal{A}_{\text{Diag}}) = \text{Acc}(\mathcal{A}'_{\text{Diag}})$ but $r(\text{Diag}', I) \leq t(\text{size}(I))$ for all $I \in L$. So, in the previous, P can be substituted by Diag' and one gets:

$$\text{Diag accepts } I(\Upsilon, \mathcal{A}'_{\text{Diag}}) \text{ iff } \text{Diag}' \text{ rejects } I(\Upsilon, \mathcal{A}_{\text{Diag}})$$

Because of the equivalence of Diag and Diag':

$$\text{Diag accepts } I(\Upsilon, \mathcal{A}'_{\text{Diag}}) \text{ iff Diag rejects } I(\Upsilon, \mathcal{A}'_{\text{Diag}})$$

This is a contradiction. □

3.3 Some Useful Definitions

In this section, some useful definitions will be given. They permit to formulate the later corollaries, lemmas and theorems more succinctly.

The following definition a set of certain vocabularies. For these vocabularies Υ, it is guaranteed that the background of an Υ is only the obigatory (logic) one and therefore it is finite. There are some propositions that can only be proven in this special case.

Definition 3.16.
1. $\mathrm{voc}_{\mathrm{finite}} := \{\Upsilon : \Upsilon \in \mathit{voc}$ and Υ contains only $\vee$, $\wedge$, $\neg$, true, false as static functions$\}$

2. For $\Upsilon \in \mathrm{voc}_{\mathrm{finite}}$, let $R_\Upsilon(T)$ be a complexity class. Then $R_{\mathrm{finite}}(T) := \bigcup_{\Upsilon \in \mathrm{voc}_{\mathrm{finite}}} R_\Upsilon(T)$.

Now, define certain types of vocabularies. These types of vocabularies will appear rather often in the following. The reason is that for some proofs, it is required that the considered vocabularies contain certain function names.

Definition 3.17. Let $\Upsilon \in \mathrm{voc}$.

1. Υ is called *order-solvent* if $\Upsilon \subset \{\mathrm{firstActive}, \mathrm{lastActive}, \mathrm{SuccActive}\}$ such that firstActive and lastActive are dynamic nullary function names and SuccActive is a dynamic unary function name.

2. Υ is called *copy-solvent* if $\{\mathrm{copy}, \mathrm{invcopy}\} \subset \Upsilon$ such that copy and invcopy are unary dynamic function names.

3. Υ is called *graph-solvent* if $\{E, E^{-1}, \mathrm{last}, \mathrm{lastVertex}\} \subset \Upsilon$ such that E and E^{-1} are 2-ary dynamic function names (marked relational), last is a nullary dynamic function name.

Now, define some sets of structures which will be used later.

Definition 3.18.
1. Let Υ be an order-solvent vocabulary. $\mathrm{ActiveOrder}_\Upsilon := \{I : I$ is an Υ-structure of finite size, $\mathrm{SuccActive}^I$ is a successor function on $\mathrm{Active}(I)$ which orders $\mathrm{Active}(I)$ totally linearly, $\mathrm{firstActive}^I$ gives the first element and $\mathrm{lastActive}^I$ gives the last element, $\mathrm{Mode}^I = \mathrm{true}, \mathrm{Output}^I = \mathrm{false}\}$.

2. Let Υ be a copy-solvent vocabulary. $\mathrm{EmptyCopy}_\Upsilon := \{I : I$ is an Υ-structure of finite size, $\mathrm{copy}^I(x) = \mathrm{undef}$ for all elements x of the universe of I, $\mathrm{invcopy}^I(x) = \mathrm{undef}$ for all elements x of the universe of I and $\mathrm{Mode}^I = \mathrm{true}, \mathrm{Output}^I = \mathrm{false}\}$.

3. Let Υ be a graph-solvent vocabulary. $\text{EmptyGraph}_\Upsilon := \{I : I$ is an Υ-structure of finite size, $E(x,y) = \text{undef}$, $E^{-1}(x,y) = \text{undef}$ for all elements x, y of the universe of I, $\text{last}^I = \text{undef}$ and $\text{Mode}^I = \text{true}, \text{Output}^I = \text{false}\}$.

The following definition gives the possibility to consider only a part of the elements of a complexity class.

Definition 3.19. Let $\Upsilon \in \text{voc}$ and $L \subseteq \{I : I$ is an Υ-structure of finite size such that $\text{Mode}^I = \text{true}, \text{Output}^I = \text{false}\}$ and $R_\Upsilon(T)$ be a complexity class. Then $R_\Upsilon(T) \restriction L := R_\Upsilon(T) \cap \{(\tilde{L}, L) : \tilde{L} \subseteq L\}$.

If $I \in \text{ActiveOrder}_\Upsilon$ for some $\Upsilon \in \text{voc}$ then even the following inequation holds: $|\text{Active}(I)| \leq \text{size}(I)$

W.l.o.g., one can assume that every state J in a run of an ASM-program over an order-solvent $\Upsilon \in \text{voc}$ on a structure $I \in \text{ActiveOrder}_\Upsilon$ has the following property:
SuccActive^J is a successor function on $\text{Active}(J)$ which orders $\text{Active}(J)$ totally linearly, firstActive^J gives the first element and lastActive^I gives the last element Else, introduce additional function names executing the role of SuccActive^I, firstActive, lastActive and introduce instructions which causes that if an element becomes active then it becomes the successor of the last active element and then the element itself becomes the last active element. If an element is not active anymore then remove it from the order.

Chapter 4

Time Complexity

There exists a mapping f that maps every structure of finite size (of an arbitrary vocabulary) to an intial structure such that for every structure $f(I)$, without the successor function on the active part and apart Mode and Output, is isomorphic to I. More precisely, there are infinitely many mappings with this property. Fix one such mapping and call it init .
Let L_1, L_2 be sets of Υ-structures. Then $\text{init}(L_1) := \{\text{init}(I) : I \in L_1\}$ and $\text{init}((L_1, L_2)) := (\text{init}(L_1), \text{init}(L_2))$.
W.l.o.g., one can assume that the vocabulary of the considered structure I does not contain the unary function name Succ-active and the nullary function name first-active and that init(I) is a structure over $\Upsilon \cup \{\text{Succ} - \text{active}, \text{first} - \text{active}\}$.
For some of the following theorems it will be necessary that the considered structures are initial. Because of the previous, this is not a big restriction. The reason for needing it is that deterministic and rule-non-deterministic programs can only deal with those elements of a structure which are accessible via a ground term. But for the proof of some propositions it is necessary to be able to access all elements of the active part (e.g., for generating a copy). For initial structures, it is rather simple to access every element of the active part (because every is the n-th successor of the first active element for some $n \in \mathbb{N}$).
In the case that active elements can not be accessed via active elements, only via a sequence of background elements this might have some curious effects. E.g., consider the following set of structures over the vocabulary $\Upsilon_{\text{inf}} = \{f, \text{Succ}, 0\} \cup \{\vee, \wedge, \neg, \text{true}, \text{false}, \text{undef}, \text{Mode}, \text{Output}\} \in \text{voc}$ where:

- f is a unary function name.
- Succ is a unary function name.
- 0 is a nullary function name.
- $\vee, \wedge, \neg, \text{true}, \text{false}, \text{undef}, \text{Mode}, \text{Output}$ have the already described arities.
- $\text{Succ}, 0, \vee, \wedge, \neg, \text{true}, \text{false}, \text{undef}$ are static function names.

- $f, \mathrm{Mode}, \mathrm{Output}$ are not static.

Now, define $L_{\inf}$ as follows:
$L_{\inf} := \{I : I$ is an $\Upsilon_{\inf}$-structure of finite size with
$(\mathbb{N} \cup \{\mathrm{true}, \mathrm{false}\}, 0, \mathrm{Succ}, \vee, \wedge, \neg, \mathrm{true}, \mathrm{false})$ as background structure and there exists at least one element x in the universe of I such that $f^I(x) \neq \mathrm{undef}$ and $\mathrm{Mode}^I = \mathrm{true}\}$
Consider the sequence $(I_n)_{n\in\mathbb{N}}$ where I_n is the following structure:
$L \in L_{\inf}$ and $f^{I_n}(n) = 0$ (and therefore $f^{I_n}(n) \neq \mathrm{undef}$) and $f^{I_n}(m) = \mathrm{undef}$ for $m \neq n$, $\mathrm{Mode}^{I_n} = \mathrm{true}, \mathrm{Output}^{I_n} = \mathrm{false}$.
$\mathrm{size}(I_n) = 1$ for all $n \in \mathbb{N}$. But the active element in I_n can only be accessed as $\mathrm{Succ}^n(0)$ and therefore it can not be accessed directly nor by a run of limited length.
In $\mathrm{init}(I_n)$ the active element can be directly accessed as $\mathrm{first} - \mathrm{active}$.
Following to the previous remark, one restrict attention to initial structures as input structures but the accessibility is not necessary for all propositions.

4.1 Deterministic Time Measures

There exist two possible time measures for ASMs: via *macrosteps* and via *microsteps*. These two different definitions lead to partly different results concerning complexity theory such that one has to consider them separately.

Definition 4.1. Given a (deterministic) ASM-program P over $\Upsilon \in \mathrm{voc}$ and an Υ-structure $I \in \mathrm{Inputstates}(P)$. Let $s_0 s_1 ... s_k$, $I = s_0$, be the run of P on I as a sequence of states and s_k is a final state. Then k is the (deterministic) macro time $\mathrm{TIME}^{\mathrm{macro}}(P, I)$ needed by P for the computation on I.

One could also define the macro time as the number of macrosteps in the run of P on I where one execution of the entire program P counts as one macrostep.

Definition 4.2. The (deterministic) macro time needed by an ASM-program P over Υ for structures from $L \subseteq \mathrm{Inputstates}(P)$ of size n is given by

$$\mathrm{TIME}^{\mathrm{macro}}(P, L, n) := \sup\{\mathrm{TIME}^{\mathrm{macro}}(P, I) : I \in L \text{ such that } \mathrm{size}(I) = n\}$$

Another possiblity to measure the time complexity, is to count every function evaluation and every pass of control. This is done by microsteps. In [1], this is called "honest evaluation time".

Definition 4.3. Given a (deterministic) syntactical entity e and a state A such that the vocabulary of e is a subset of the vocabulary of A. The *honest evaluation time* $\mathrm{HET}(e, A)$ for evaluating e at A is defined inductively:

- If e is a variable, then $\mathrm{HET}(e, A) = 1$.
- If e is a term $f(t_1, ..., t_r)$, then $\mathrm{HET}(e, A) = 1 + \sum_{i=1}^{r} \mathrm{HET}(t_i, A)$
- If e is an atomic instruction $f(t_1, ..., t_r) := t_0$, then $\mathrm{HET}(e, A) = 1 + \sum_{i=0}^{r} \mathrm{HET}(t_i, A)$
- If e is a block rule with constituents $R_1, ..., R_k$, then $\mathrm{HET}(e, A) = 1 + \sum_{i=1}^{k} \mathrm{HET}(R_i, A)$
- Suppose that e is a clause "if g then R". If g holds at A, then $\mathrm{HET}(e, A) = 1 + \mathrm{HET}(g, A) + \mathrm{HET}(R, A)$; otherwise $\mathrm{HET}(e, A) = 1 + \mathrm{HET}(g, A)$
- If e is an import rule with body R, then $\mathrm{HET}(e, A) = 1 + \mathrm{HET}(R, A)$.

Definition 4.4. Given a (deterministic) ASM-program P over $\Upsilon \in$ voc and an Υ-structure $I \in \mathrm{Inputstates}(P)$. Let $s_0 s_1 ... s_k$, $I = s_0$, be the run of P on I as a sequence of states and s_k is a final state. Then the (deterministic) micro time is $\mathrm{TIME}^{\mathrm{micro}}(P, I) = \sum_{i=0}^{k-1} \mathrm{HET}(P, s_i)$.

Definition 4.5. The (deterministic) micro time needed by an ASM-program P over Υ for structures from $L \subseteq \mathrm{Inputstates}(P)$ of size n is given by

$$\mathrm{TIME}^{\mathrm{micro}}(P, L, n) := \sup\{\mathrm{TIME}^{\mathrm{micro}}(P, I) : I \in L \text{ such that size}(I) = n\}$$

4.2 Non-Deterministic Time Measures

As in the case of determinism, there are two possible time measures for ASMs: via macrosteps and via microsteps. Two kinds of non-deterministic ASM-programs have to be considered: (1) the deterministic rules together with the guess-element rule (element-non-determinism) and (2) the deterministic rules together with only the guess-rule rule (rule-non-determinism).
Non-deterministic complexity measures and complexity classes (time and space) will always have an index $i \in \{\mathrm{el}, \mathrm{rule}\}$. This indicates the kind of non-determinism currently considered. The index *el* indicates element-non-determinism, the index *rule* indicates rule-non-determinism.

Definition 4.6. Given a non-deterministic ASM-program P (element-non-deterministic or rule-non-deterministic) and an Υ-structure $I \in \mathrm{Inputstates}(P)$. Let $M = \{s_0 s_1 ... s_k : I = s_0$ and $s_0 s_1 ... s_k$ is a run of P on I as a sequence of states and s_k is a final state$\}$. Then the non-deterministic macro time is $\mathrm{NTIME}_i^{\mathrm{macro}}(P, I) = \sup\{\mathrm{length}(Y) : Y \in M\}$.

Definition 4.7. For $i \in \{\text{el}, \text{rule}\}$ the non-deterministic macro time needed by an ASM-program P over Υ for structures from $L \subseteq \text{Inputstates}(P)$ of size n is given by

$$\text{NTIME}_i^{\text{macro}}(P, L, n) := \sup\{\text{NTIME}_i^{\text{macro}}(P, I) : I \in L \text{ such that } \text{size}(I) = n\}$$

Definition 4.8. Given a syntactical entity e (composed from deterministic rules and the guess-element rule) with n guess-element-rules, a state A such that the vocabulary of e is a subset of the vocabulary of A and n living elements $a_1, ..., a_n$ from A. The *non-deterministic honest evaluation time* $\text{NHET}_{\text{el}}(e, A, a_1, ..., a_n)$ for evaluating e at A with guessing a_j in the j-th guess-element-rule is defined inductively:

- If e is a variable, then $\text{NHET}_{\text{el}}(e, A) = 1$.
- If e is a term $f(t_1, ..., t_r)$, then $\text{NHET}_{\text{el}}(e, A) = 1 + \sum_{i=1}^{r} \text{NHET}_{\text{el}}(t_i, A)$
- If e is an atomic instruction $f(t_1, ..., t_r) := t_0$, then $\text{NHET}_{\text{el}}(e, A) = 1 + \sum_{i=0}^{r} \text{NHET}_{\text{el}}(t_i, A)$
- If e is a block rule with constituents $R_1, ..., R_k$, then $\text{NHET}_{\text{el}}(e, A, a_1, ..., a_n) = 1 + \sum_{i=1}^{k} \text{NHET}_{\text{el}}(R_i, A, a_1^i, ..., a_{n_i}^i)$ where R_i contains n_i guess-element-rules and $a_1^i, ..., a_{n_i}^i$ are the guessed elements (received from $a_1, ..., a_n$).
- Suppose that e is a clause "if g then R". If g holds at A, then $\text{NHET}_{\text{el}}(e, A, a_1, ..., a_n) = 1 + \text{NHET}_{\text{el}}(g, A) + \text{NHET}_{\text{el}}(R, A, a_1, ..., a_n)$; otherwise $\text{NHET}_{\text{el}}(e, A, a_1, ..., a_n) = 1 + \text{NHET}_{\text{el}}(g, A)$
- If e is an import rule with body R, then $\text{NHET}_{\text{el}}(e, A, a_1, ..., a_n) = 1 + \text{NHET}_{\text{el}}(R, A, a_1, ..., a_n)$.
- If e is a guess-element rule where v is the head variable, $g(v)$ is the guard and R is the body.
  $\text{NHET}_{\text{el}}(e, A, a_1, ..., a_n) = 1 + \text{NHET}_{\text{el}}(R[v/a_1], A, a_2, ..., a_n)$ where $R[v/a_1]$ for an $a_1 \in M$ is obtained by replacing all occurences of v in R by a_1.

Note: If e is a variable, a term or an atomic instruction, then e does not contain any guess-element rule and therefore, the honest non-deterministic evaluation time does not depend on any guessed elements (of the structure).

Definition 4.9. Given a syntactical entity e (where element-non-determinism is not used, only rule-non-determinism) and a state A of sufficiently rich vocabulary. The *non-deterministic honest evaluation time* $\text{NHET}_{\text{rule}}(e, A)$ for evaluating e at A is defined analogously to the one in definition 4.8 where instead of the guessed elements from the universe, elements from $\{1, 2\}$ are used where the number gives

the guessed rule. The last point has to be replaced by the following:
If e is a guess-rule rule with rules R_1 and R_2
$\mathrm{NHET}_{\mathrm{rule}}(e, A, 1, ..., a_n) = 1 + \mathrm{NHET}_{\mathrm{rule}}(R_1, A, a_2, ..., a_n)$
$\mathrm{NHET}_{\mathrm{rule}}(e, A, 2, ..., a_n) = 1 + \mathrm{NHET}_{\mathrm{rule}}(R_2, A, a_2, ..., a_n)$

Definition 4.10. Given a non-deterministic ASM-program P (element-non-deterministic or rule-non-deterministic) and an Υ-structure $I \in \mathrm{Inputstates}(P)$. Let $M = \{s_0 s_1 ... s_k : I = s_0$ and $s_0 s_1 ... s_k$ is a run of P on I as a sequence of states and s_k is a final state$\}$. Then the non-deterministic micro time is $\mathrm{NTIME}_i^{\mathrm{micro}}(P, I) = \sup\{\sum_{j=0}^{k-1} \mathrm{NHET}_i(P, s_j, a_1^j, ..., a_n^j) : s_0 s_1 ... s_k \in M$ and s_{i+1} is the successor of s_i with respect to P and the guesses $a_1^i, ..., a_n^i \in \mathrm{living}(s_i)\}$.

Definition 4.11. The non-deterministic micro time needed by an ASM-program P over Υ for structures from $L \subseteq \mathrm{Inputstates}(P)$ of size n is given by

$$\mathrm{NTIME}_i^{\mathrm{micro}}(P, L, n) := \sup\{\mathrm{NTIME}_i^{\mathrm{micro}}(P, I) : I \in L \text{ such that } \mathrm{size}(I) = n\}$$

4.3 Macrostep-Complexity-Classes

Definition 4.12. Fix $\Upsilon \in \mathrm{voc}$.
$\mathrm{TIME}_\Upsilon^{\mathrm{macro}}(f(n)) := \{(L_1, L_2) : L_1, L_2$ are sets of Υ-structures of finite size, $L_1 \subseteq L_2$, and there exists a (deterministic) ASM $\mathcal{A} = (P, L_2)$ over Υ such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{TIME}^{\mathrm{macro}}(P, \mathrm{size}(I)) \leq f(\mathrm{size}(I))$ for all $I \in L_2\}$

Definition 4.13. $\mathrm{TIME}^{\mathrm{macro}}(f(n)) := \bigcup_{\Upsilon \in \mathrm{voc}} \mathrm{TIME}_\Upsilon^{\mathrm{macro}}(f(n))$

Definition 4.14. Fix $\Upsilon \in \mathrm{voc}$. Let $i \in \{\mathrm{el}, \mathrm{rule}\}$.
$\mathrm{NTIME}_{i,\Upsilon}^{\mathrm{macro}}(f(n)) := \{(L_1, L_2) : L_1, L_2$ are sets of Υ-structures of finite size, $L_1 \subseteq L_2$, and there exists a non-deterministic ASM $\mathcal{A} = (P, L_2)$ of kind i over Υ such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{NTIME}^{\mathrm{macro}}(P, \mathrm{size}(I)) \leq f(\mathrm{size}(I))$ for all $I \in L_2\}$

Definition 4.15. Let $i \in \{\mathrm{el}, \mathrm{rule}\}$.
$\mathrm{NTIME}_i^{\mathrm{macro}}(f(n)) := \bigcup_{\Upsilon \in \mathrm{voc}} \mathrm{NTIME}_{i,\Upsilon}^{\mathrm{macro}}(f(n))$

Theorem 4.16 (Speed-Up-Theorem). *If $(L_1, L_2) \in \mathrm{TIME}^{\mathrm{macro}}(f(n))$, then $(L_1, L_2) \in \mathrm{TIME}^{\mathrm{macro}}(c \cdot f(n))$ for every $c > 0$.*

Proof. Let $(L_1, L_2) \in \mathrm{TIME}^{\mathrm{macro}}(f(n))$.
For $c \geq 1$, $L \in \mathrm{TIME}^{\mathrm{macro}}(c \cdot f(n))$ is a trivial statement.
Let $0 < c < 1$. Because $(L_1, L_2) \in \mathrm{TIME}^{\mathrm{macro}}(f(n))$, there exists an ASM $\mathcal{A} = (P, L_2)$ such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{TIME}^{\mathrm{macro}}(P, \mathrm{size}(I)) \leq f(\mathrm{size}(I))$ for all $I \in L_2$. Define $k := \lceil c^{-1} \rceil$. Build a program P^k that executes k turns and

therefore k macrosteps of P at once. The rough idea for this is to determine a sequence of k macrosteps of P and write the result into a program.
Consider two (deterministic) rules in normal form according to lemma 2.5. In the following, the composition of two rules R_1 and R_2, $R_1 \circ R_2$, is defined. Applying $R_1 \circ R_2$ to an Υ-structure I leads to the same state as first applying R_1 to I and then applying R_2 to the result.
The rough idea for constructing $R_1 \circ R_2$ from R_1 and R_2 is the following:
A set of instructions has to be applied to R_1 and R_2 until it is not possible anymore. With these rules, one goes through R_1 and R_2 and considers their influences on each other. These influences are directed from R_1 to R_2 because R_1 is executed first and R_2 can not influence R_1. If in R_1, a term t is updated and this term is a subterm of the right side in any update of R_2 then the updated value of t has to be used in R_2 at these positions. If t occurs as a real subterm on the left side of an update in R_2 then the same has to be done. If t itself occurs on the left side of an update in R_2 then the update excuted by R_1 does not interest anymore. The concrete formulation of this idea is not that direct because the updates are situated inside of if-clauses and therefore, not all updates appearing in R_1 are executed and the guards of the if-clauses in R_2 have to be modified just like the updates. Furthermore, the updates in R_1 are connected among each other and so are those in R_2.
One has to be careful with inconsistencies in R_1 and to be careful not to introduce any inconstencies, which originally did not exist, into the resulting program.
Concretely or more precisely, this will be described now.
Apply each of the following instructions to R_1 and R_2 until it is not possible anymore in the order indicated by the numbering:

1. If R_1 is a rule of the form `import` v_1; $\tilde{R}_1$; `endimport` and R_2 is a rule of the form `import` v_2; $\tilde{R}_2$; `endimport` where, w.l.o.g. v_1 and v_2 are distinct variables, v_1 does not occur in $\tilde{R}_2$ and v_2 does not occur in $\tilde{R}_1$ (else rename the bound variables) then $R_1 \circ R_2$ is the following rule:

   `import` v_1
       `import` v_2
           $\tilde{R}_1 \circ \tilde{R}_2$
       `endimport`
   `endimport`

2. If R_1 is a rule of the form `import` v_1; $\tilde{R}_1$; `endimport` and R_2 is not an import-rule (it is a block of if-clauses) where,w.l.o.g., v_1 does not occur in R_1 (else rename v_1 respectively) then $R_1 \circ R_2$ is the following rule:

   `import` v_1
       $\tilde{R}_1 \circ R_2$
   `endimport`

3. If R_2 is a rule of the form `import` v_2`;` $\tilde{R}_2$`;` `endimport` and R_1 is not an import-rule where, w.l.o.g. v_2 does not occur in R_1 (else rename) then $R_1 \circ R_2$ is the following rule:

   `import` v_2
   $\quad R_1 \circ \tilde{R}_2$
   `endimport`

4. Now, the problem is reduced to two blocks of if-clauses where the body of each if-clause consists of only one update-rule. I.e., starting from now, R_1 and R_2 are assumed to be sequences of if-clauses.
   Now, do the following:
   For every $f \in \Upsilon_{\text{dyn}}$ do the following: Let `if` g_1 `then` $f(s_1^1, ..., s_r^1) := s_0^r$ `endif; ...; if` g_n `then` $f(s_1^n, ..., s_r^n) := s_0^n$ `endif` be all if-clauses in R_1 with an update-rule causing an update of a location with f. Then build the sequence of all if-clauses of the follwing form for $a \neq b$, $a, b \in \{1, ..., n\}$:

   `if` $g_a \wedge g_b \wedge \bigwedge_{i \in \{1,...,r\}} s_i^a = s_i^b \wedge s_0^a \neq s_0^b$ `then`
   $\quad$`block`
   $\quad\quad f(s_1^a, ..., s_r^a) := s_0^a$
   $\quad\quad f(s_1^b, ..., s_r^b) := s_0^r$
   $\quad$`endblock`
   `endif`

   Let P be the sequence of these if-clauses and for all $f \in \Upsilon_{\text{dyn}}$ transformed in normal form. Then $R_1 \circ R_2$ is equal to

   `block`
   $\quad P$
   $\quad R_1 \circ' R_2$
   `endblock`

   where for two rules Q_1, Q_2, each consisting of a block of if-clauses with only one update-rule as body and wth distinct variables, $Q_1 \circ' Q_2$ is obtained by applying each of the following instructions until it is not possible anymore and in the order indicated by the numbering:

   (a) Suppose Q_2 contains an update-rule $s := t$ inside an if-clause and Q_1 contains one or more if clauses `if` g_1 `then` $x := y$ `endif; ...; if` g_m `then` $x := y$ `endif` (which is equivalent to `if` $\bigvee_{i \in \{1,...,m\}} g_i$ `then` $x := y$ `endif`).
   If t contains a subterm x, then replace $s := t$ (in Q_2) by the following:

   `if` $\bigvee_{i \in \{1,...,m\}} g_i$ `then` $s := t[x/y]$ `endif`

Do this for all those updates $s := t$ satisfying the described conditions. This leads to Q_2'.
Transform Q_2' into normal form (this leads to Q_2''). $Q_1 \circ' Q_2$ is equal to $Q_1' \circ' Q_2''$.

(b) Suppose Q_2 contains an update-rule $s := t$ inside an if-clause and Q_1 contains one or more if clauses `if` g_1 `then` $x := y$ `endif; ...; if` g_m `then` $x := y$ `endif` (which is equivalent to `if` $\bigvee_{i \in \{1,\ldots,m\}} g_i$ `then` $x := y$ `endif`).
If s contains a real subterm x (i.e., x is a subterm of s, x and s are not syntactically equal), then replace $s := t$ (in Q_2) by the following:

`if` $\bigvee_{i \in \{1,\ldots,m\}} g_i$ `then` $s[x/y] := t$ `endif`

Do this for all those updates $s := t$ satisfying the described conditions. This leads to Q_2'.
Transform Q_2' into normal form (this leads to Q_2''). $Q_1 \circ' Q_2$ is equal to $Q_1' \circ' Q_2''$.

(c) Suppose Q_2 contains an if-clause `if` h `then` Q `endif`.
If g contains a term x updated within an if-clause with y (as above, w.l.o.g., the case `if g then` $x := y$ is considered), then replace `if` h `then` Q `endif` in Q_2 by the following:

```
if g then
  block
    if h[x/y] then Q̃ endif
    x := y
  endblock
endif
if ¬g then
  if h then Q endif
endif
```

Transformed into normal form, this leads to Q_2'. $Q_1 \circ' Q_2$ is equal to $Q_1 \circ' Q_2'$.

(d) If there is an if-clause `if` g `then` Mode := false `endif` in Q_1 (w.l.o.g., one can suppose that there is at most one such clause because the sequence `if` g_1 `then` Mode := false `endif; ...; if` g_n `then` Mode := false `endif` is equivalent to `if` $\bigvee_{i \in \{1,\ldots,n\}} g_i$ `then` Mode := false `endif`) then it is removed from Q_1 and added to the block of if-clauses in Q_2.
If, in addition to this, there is an if-clause `if` h `then` Output := x `endif` ($x \in$ Boole) then remove it from Q_1 and write `if` $g \wedge h$ `then` Output := x `endif` into Q_2.
This leads to Q_1' and R_2'. $Q_1 \circ' Q_2$ is equal to $Q_1' \circ' Q_2'$.

(e) The next instruction is introduced for avoiding to introduce additional inconsistencies. Consider an if-clause `if` g_1 `then` $f(s_1, ..., s_r) := s_0$ `endif` in Q_1 and all if-clauses in Q_2 containing an updated updating locations with function name $f \in \Upsilon_{dyn} - \{\text{Mode}, \text{Output}\}$ `if` g_2^1 `then` $f(t_1^1, ..., t_r^1) := t_0^1$ `endif ;...; if` g_2^n `then` $f(t_1^n, ..., t_r^n) := t_0^n$ `endif` $f(s_1, ..., s_r)$ and $f(t_1^i, ..., t_r^i)$ are syntactically not equal ($i \in \{1, ..., n\}$). Replace the if-clause in Q_1 by

`if` $g_1 \wedge g_2 \wedge \bigwedge_{i\in\{1,...,n\}} \bigvee_{j\in\{1,...,r\}} s_j^i \neq t_j^i \wedge \bigwedge_{i\in\{1,...,n\}} s_0^i \neq t_0^i$
`then` $f(s_1, ..., s_r) := s_0$ `endif`

This leads to Q_1'. Then $Q_1 \circ' Q_2$ is equal to $Q_1' \circ'' Q_2$ where for two rules S_1, S_2, each consisting of a block of if-clauses with only one update-rule as body, $S_1 \circ'' S_2$ is obtained by applying each of the following instructions until it is not possible anymore and in the order indicated by the numbering:

i. Suppose S_2 contains an if-clause `if` g_2 `then` $s := t$ `endif` and S_1 contains an update `if` g_1 `then` $s := \tilde{t}$ `endif`. Then replace the if-clause in S_2 by the following:

`if` $g_1 \wedge g_2$ `then` $s := t$ `endif`
`if` $g_1 \wedge \neg g_2$ `then` $s := \tilde{t}$ `endif`
`if` $\neg g_1 \wedge g_2$ `then` $s := t$ `endif`

Delete the considered if-clause in S_2. Both resulting rules (S_1' and S_2' are in normal form. Then $S_1 \circ'' S_2$ is equal to $S_1' \circ'' S_2'$ This treats the case that each of the rules, S_1 and S_2, contains only one of these clauses. This can be extended the case that one rule or both rules contain more than one such if-clause.

ii. This case is reached if none of the preceding cases holds then $S_1 \circ'' S_2$ is equal to the sequence of S_1 and S_2. I.e., $S_1 \circ'' S_2$ is the following rule:

`block`
  S_1
  S_2
`endblock`

This process always terminates.
Because of 4., inconsistencies appearing in R_1 are preserved. A program produces an inconsistent update set for a structure I iff there are two if-clauses `if` g_1 `then` $f(s_1, ..., s_r) := s_0$ `endif` and `if` g_2 `then` $f(t_1, ..., t_r) := t_0$ `endif` such that g_1 and g_2 hold in I, s_i and t_i ($i \in \{1, ..., r\}$) evaluate to the same element in I, s_0 and t_0 evaluate to different elements in I. Because of the guard of the if-clauses introduced in 4.., these two contradicting updates do also appear in the update

set of the resulting program. Therefore, if R_1 is inconsistent then $R_1 \circ R_2$ is inconsistent.
Because of 4. (d), no new inconsistencies are introduced. The updates coming from R_1 are only executed if they do not collide with those from R_2. If there is such a collision then only the update coming from R_2 is executed because the one of R_1 is overwritten.
The update set resulting from $R_1 \circ R_2$ is inconsistent iff the update set resulting from R_1 is inconsistent or the update set resulting from R_2 is inconsistent.
If via the execution of R_1, a final state is reached then it is also reached via $R_1 \circ R_2$ with the same result. This is ensured by 4..
By induction on R_1 and R_2, one can conclude that $R_1 \circ R_2$ has the intended poperty.
Now, define P^n by induction on $n \in \mathbb{N} - \{0\}$.

- $P^1 := P$
- $P^{n+1} := P \circ P^n$

For all $I \in L_2$, I is accepted by P iff I is accepted by P^k and because $\mathrm{TIME}^{\mathrm{macro}}(P, \mathrm{size}(I)) \leq f(\mathrm{size}(I))$ for all $I \in L_2$, one gets $\mathrm{TIME}^{\mathrm{macro}}(P^k, \mathrm{size}(I)) \leq k^{-1} \cdot f(\mathrm{size}(I))$.
Because $k^{-1} \leq c$, for all structures I $\mathrm{TIME}^{\mathrm{macro}}(P^k, \mathrm{size}(I)) \leq c \cdot f(\mathrm{size}(I))$ for all $I \in L_2$.
Therefore, the ASM (P^k, L_2) witnesses that $L \in \mathrm{TIME}^{\mathrm{macro}}(c \cdot f(n))$. □

Definition 4.17. Let $\Upsilon = \{\mathrm{Mode}, \mathrm{Output}, \mathrm{true}, \mathrm{false}, \mathrm{undef}, \mathrm{Boole}, \neg, \vee, \wedge, =\} \cup \{\mathrm{firstActive}, \mathrm{lastActive}, \mathrm{SuccActive}\} \in \mathrm{voc}$ such that firstActive and lastActive are dynamic nullary function names and SuccActive is a dynamic unary function name. The rest of the vocabulary (the obligatory part) is as already described. Let $\mathrm{Order} := \{I : I$ is an Υ-structure of finite size, $\mathrm{SuccActive}^I$ is a successor function in $\mathrm{Active}(I)$ which orders $\mathrm{Active}(I)$ totally linearly, $\mathrm{firstActive}^I$ gives the first element and $\mathrm{lastActive}^I$ gives the last element, $\mathrm{Mode}^I = \mathrm{true}, \mathrm{Output}^I = \mathrm{false}\}$.
A function $f : \mathbb{N} \to \mathbb{N}$ is called *macro time constructible* if there is an (deterministic) ASM $\mathcal{A} = (P, \mathrm{Order})$ such that for every $I \in \mathrm{Order}$, $\mathrm{TIME}^{\mathrm{macro}}(P, I) = f(|\mathrm{Active}(I)|)$.

Examples for macrotime constructable functions are all polynomial functions, $f(n) = 2^n$,... . If $f, g : \mathbb{N} \to \mathbb{N}$ are macro time constructable functions then $f + g$, $f \cdot g$, $f \circ g$ are also macro time constructable.

Lemma 4.18. *Let $\Upsilon \in voc$ be an order-solvent vocabulary and f be a macro time constructible function.*
Let $\mathcal{A} = (P, \mathrm{Input})$ be a (deterministic) ASM such that $\mathrm{Input} \subseteq \mathrm{ActiveOrder}_\Upsilon$*,* $\mathrm{TIME}^{\mathrm{macro}}(P, I) \leq f(\mathrm{size}(I))$ *for all accepted structures* $I \in \mathrm{Input}$.

Then there exists a (deterministic) ASM $\mathcal{A}' = (P', \text{Input})$ *such that* $\text{Acc}(\mathcal{A}') = \text{Acc}(\mathcal{A})$ *and* $\text{TIME}^{\text{macro}}(P', I) \leq \sum_{g\in\Upsilon_{\text{dyn}}} |\text{Active}(I)|^{\text{arity}(g)} + f(\text{size}(I))$ *for all* $I \in \text{Input}$.

Note: $\mathcal{A}'$ halts on every input structure but $\mathcal{A}$ does not need to do so.

Proof. Let $\Upsilon \in voc$ be an order-solvent vocabulary, f be a macro time constructible function and $\mathcal{A} = (P, \text{Input})$ be a (deterministic) ASM over Υ such that $\text{TIME}^{\text{macro}}(P, I) \leq f(\text{size}(I))$ for all accepted structures $I \in \text{Input}$.
Because f is macro time constructible, an ASM $\mathcal{B} = (Q, \text{Order})$ exists that does exactly $f(|\text{Active}(J)|)$ macrosteps on every structure $J \in \text{Order}$.
Choose as program P' the program which does the following: First, for all $g \in \Upsilon_{dyn}$ all arity(g)-tuples are considered in the lexicographical order induced by the interpretation of SuccActive. If g is not interpreted as undef for this tuple then a new element is imported for this, the successor of the last element is set to this new one and it becomes the last element. This leads a strucure consisting of two disjoint substructures (apart from Mode and Output): an element from Order with $f(\text{size}(I))$ active elements and I.
The first such element resp. tuple is an exception. The first and last element are set to it. The successor-function is undefined.
The above is done until no more such tuple is left. The result is a structure consisting of two disjoint substructures. The first one is the original structure (with a successor function defined on it) and the second one is an initial structure with a marked last element. The second structure contains size(I) active elements.
The rest of the program succeeds as follows: The ASM $\mathcal{A}$ is executed on the original structure and in parallel to it, the ASM $\mathcal{B}$ is executed on the second part of the structure, i.e., the initial structure with an active part of size size(I). $\mathcal{A}'$ accepts an input structure IInput iff $\mathcal{A}$ accepts it within $f(\text{size}(I))$ macrosteps. Otherwise $\mathcal{A}'$ rejects after $f(\text{size}(I))$ macrosteps. It can be detected whether $f(\text{size}(I))$ macrosteps are already done by controling whether P' has already stopped. By use of the Speed-Up-Theorem, one can conclude the lemma. □

4.4 Microstep-Complexity-Classes

Definition 4.19. Fix $\Upsilon \in$ voc.
$\text{TIME}^{\text{micro}}_{\Upsilon}(f(n)) := \{(L_1, L_2) : L_1, L_2$ are sets of Υ-structures of finite size, $L_1 \subseteq L_2$, and there exists a (deterministic) ASM $\mathcal{A} = (P, L_2)$ over Υ such that $L_1 = \text{Acc}(\mathcal{A})$ and $\text{TIME}^{\text{micro}}(P, \text{size}(I)) \leq f(\text{size}(I))$ for all $I \in L_2\}$

Definition 4.20. $\text{TIME}^{\text{micro}}(f(n)) := \bigcup_{\Upsilon\in\text{voc}} \text{TIME}^{\text{micro}}_{\Upsilon}(f(n))$

Definition 4.21. Fix $\Upsilon \in$ voc. Let $i \in \{\text{el}, \text{rule}\}$.
$\text{NTIME}^{\text{micro}}_{i,\Upsilon}(f(n)) := \{(L_1, L_2) : L_1, L_2$ are sets of Υ-structures of finite size, $L_1 \subseteq L_2$, and there exists a non-deterministic ASM $\mathcal{A} = (P, L_2)$ of kind i over

Υ such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{NTIME}^{\mathrm{micro}}(P, \mathrm{size}(I)) \leq f(\mathrm{size}(I))$ for all $I \in L_2\}$

Definition 4.22. Let $i \in \{\mathrm{el}, \mathrm{rule}\}$.
$\mathrm{NTIME}_i^{\mathrm{micro}}(f(n)) := \bigcup_{\Upsilon \in \mathrm{voc}} \mathrm{NTIME}_{i,\Upsilon}^{\mathrm{micro}}(f(n))$

As shown in [1], the Speed-Up-Theorem does not hold for micro time. In [1] another notion of size has been used. Namely, the size of a structure is defined as the number of active elements of the structure but the results still hold for the notion of size in this work. One of the main results of [1] is the following (transfered into the terms of this work):

Theorem 4.23 (Linear Time Hierarchy Theorem). *For every hierarchy-solvent Υ (i.e., in addition to the logic names and the names 0,* Succ, Mode, Output*, Υ contains at least five non-relational nullary names and at least seven non-relational unary names), there exists a constant c such that, for every positive integer d,* $\mathrm{TIME}_{\Upsilon}^{micro}(c \cdot d \cdot n) - \mathrm{TIME}_{\Upsilon}^{micro}(d \cdot n) \neq \emptyset$

The assumptions of the following definition are the same as in the definition for *macro time constructible*.

Definition 4.24. Let $\Upsilon = \{\mathrm{Mode}, \mathrm{Output}, \mathrm{true}, \mathrm{false}, \mathrm{undef}, \mathrm{Boole}, \neg, \vee, \wedge, =\} \cup \{\mathrm{firstActive}, \mathrm{lastActive}, \mathrm{SuccActive}\} \in \mathrm{voc}$ such that firstActive and lastActive are dynamic nullary function names and SuccActive is a dynamic unary function name. The rest of the vocabulary (the obligatory part) is as already described. Let Order $:= \{I : I$ is an Υ-structure of finite size, $\mathrm{SuccActive}^I$ is a successor function in $\mathrm{Active}(I)$ which orders $\mathrm{Active}(I)$ totally linearly, $\mathrm{firstActive}^I$ gives the first element and $\mathrm{lastActive}^I$ gives the last element, $\mathrm{Mode}^I = \mathrm{true}, \mathrm{Output}^I = \mathrm{false}\}$. A function $f : \mathbb{N} \to \mathbb{N}$ is called *micro time constructible* if there is an (deterministic) ASM $\mathcal{A} = (P, \mathrm{Order})$ such that for every $I \in \mathrm{Order}$, $\mathrm{TIME}^{\mathrm{micro}}(P, I) = f(|\mathrm{Active}(I)|)$.

Lemma 4.25. *Let $\Upsilon \in \mathrm{voc}$ be an order-solvent vocabulary and f be a micro time constructible function.*
Let $\mathcal{A} = (P, \mathrm{Input})$ be a (deterministic) ASM such that $\mathrm{Input} \subseteq \mathrm{ActiveOrder}_{\Upsilon}$, $\mathrm{TIME}^{\mathrm{micro}}(P, I) \leq f(\mathrm{size}(I))$ for all accepted structures $I \in \mathrm{Input}$. Then there exists a (deterministic) ASM $\mathcal{A}' = (P', \mathrm{Input})$ such that $\mathrm{Acc}(\mathcal{A}') = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{TIME}^{\mathrm{micro}}(P', I) \leq \sum_{g \in \Upsilon_{\mathrm{dyn}}} |\mathrm{Active}(I)|^{\mathrm{arity}(g)} + 3 \cdot f(\mathrm{size}(I))$ for all $I \in \mathrm{Input}$.

Note: $\mathcal{A}'$ halts on every input structure but $\mathcal{A}$ does not need to do so.

Proof. Analogous to the proof of lemma 4.18 but the parallel execution introduces the constant factor 3 because the Speed-Up-Theorem does not hold for microsteps. ☐

Because per one macrostep at least one microstep is done, the following corollary is trivial:

Corollary 4.26. *Let $f : \mathbb{N} \to \mathbb{N}$, $i \in \{\text{el}, \text{rule}\}$. Then the following two statements hold:*

1. $\text{TIME}^{\text{micro}}(f(n)) \subseteq \text{TIME}^{\text{macro}}(f(n))$
2. $\text{NTIME}_i^{\text{micro}}(f(n)) \subseteq \text{NTIME}_i^{\text{macro}}(f(n))$

4.5 Consequences Of The Diagonalization Theorem

Lemma 4.27. *Let $\Upsilon \in \text{voc}$ be a diagonalization-solvent, order-solvent and copy-solvent vocabulary. Let $f_2 : \mathbb{N} \to \mathbb{N}$ be a macro time constructible function Υ and $f_1 : \mathbb{N} \to \mathbb{N}$. If $f_2(n) \geq n^{\max_{g \in \Upsilon_{\text{dyn}}}(\text{arity}(g))}$ and $f_1 = o(f_2)$ then*

$$\begin{aligned} &(\text{TIME}_\Upsilon^{\text{macro}}(f_2(n)) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyCopy}_\Upsilon)) \\ - \;&(\text{TIME}_\Upsilon^{\text{macro}}(f_1(n)) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyCopy}_\Upsilon)) \neq \emptyset \end{aligned}$$

Proof. It will be proven that for every diagonalization-solvent, order-solvent and copy-solvent vocabulary $\Upsilon \in \text{voc}$, $\text{TIME}_\Upsilon^{\text{macro}}(f_2(n))$ allows diagonalization over $\text{TIME}_\Upsilon^{\text{macro}}(f_1(n))$ concerning $\text{ActiveOrder}_\Upsilon \cap \text{EmptyCopy}_\Upsilon$. From the diagonalizition theorem, one can then conclude that this lemma holds. In the following, an ASM $\mathcal{A}_{\text{Diag}} = (\text{Diag}, (\text{ActiveOrder}_\Upsilon \cap \text{EmptyCopy}_\Upsilon))$ will be constructed which satisfies the conditions of definition 3.14. Diag works as follows:
Every $I \in \text{ActiveOrder}_\Upsilon \cap \text{EmptyCopy}_\Upsilon$ is interpreted as an ASM $\mathcal{A}$, i.e., $I = I(\Upsilon, \mathcal{A})$ (it is assumed that I encodes an ASM, if not, I is rejected). Diag simulates the work of P on I. The simulation can be done within $c_P \cdot \text{TIME}^{\text{macro}}(P, I)$ macrosteps where c_P is a constant depending only on P. Diag accepts I iff P stops in $f_2(\text{size}(I))$ macrosteps and rejects. This is achieved as follows. As in the proof of lemma 4.18, construct a disjoint structure which is totally linearly ordered by SuccActive, firstActive, lastActive. The order contains $\text{size}(I)$ elements. In parallel to the simulation of $\mathcal{A}$, an ASM is executed on the order which satisfies the property of definition 4.17 (because f_2 is macro time constructible concerning Υ such an ASM does exist). Within one macrostep of the resulting program both programs are executed once. After $f_2(\text{size}(I))$ macrosteps, Diag stops and accepts iff P has stopped and rejected.
Because of the Speed-Up-Theorem and lemma 3.2, the disjoint structure can be produced within $(\text{size}(I))^{\max_{g \in \Upsilon_{\text{dyn}}}(\text{arity}(g))}$ macrosteps.
Because of lemmas 3.2, 3.3 and the Speed-Up-Theorem, Diag can be constructued such that the number of macrosteps, Diag does on a structure $I \in \text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon$, is limited by $f_2(\text{size}(I))$ (and therefore Diag halts on every element of $\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon$).
For all ASM-programs P', it is true that $f_2(n) \geq c_{P'} \cdot f_1(n)$ for all but finitely

many $n \in \mathbb{N}$ (because $f_2 = o(f_1)$). So, for all but finitely many input structures, Diag simulates the work of P on I completely in $f_2(\mathrm{size}(I))$ macrosteps. So, $I(\Upsilon, \mathcal{A}) \in \mathrm{Acc}(\mathrm{Diag})$ iff $I(\Upsilon, \mathcal{A}) \notin \mathrm{Acc}(\mathcal{A})$. By use of the diagonalization theorem, one can conclude the lemma. $\square$

The following corollary is an obvious consequence of the preceding lemma:

Corollary 4.28. *Let* $\Upsilon \in \mathrm{voc}$ *be a diagonalization-solvent, order-solvent and copy-solvent vocabulary. Let* $f_2 : \mathbb{N} \to \mathbb{N}$ *be a macro time constructible function* Υ *and* $f_1 : \mathbb{N} \to \mathbb{N}$. *If* $f_2(n) \geq n^{\max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g))}$ *and* $f_1 = o(f_2)$ *then* $\mathrm{TIME}_{\Upsilon}^{\mathrm{macro}}(f_2(n)) - \mathrm{TIME}_{\Upsilon}^{\mathrm{macro}} \neq \emptyset$.

Lemma 4.29. *Let* $\Upsilon \in \mathrm{voc}$ *be a diagonalization-solvent, order-solvent and copy-solvent vocabulary. Let* $f_2 : \mathbb{N} \to \mathbb{N}$ *be a micro time constructible function and* $f_1 : \mathbb{N} \to \mathbb{N}$. *If* $f_2(n) \geq (|\Upsilon_{\mathrm{dyn}}| \cdot \max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g) + 1) \cdot n)^{\max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g))+1}$ *and* $f_1 = o(f_2)$ *then*
$(\mathrm{TIME}_{\Upsilon}^{\mathrm{micro}}(3 \cdot f_2(n)) \restriction (\mathrm{ActiveOrder}_{\Upsilon} \cap \mathrm{EmptyGraph}_{\Upsilon})) - (\mathrm{TIME}_{\Upsilon}^{\mathrm{micro}}(f_1(n)) \restriction (\mathrm{ActiveOrder}_{\Upsilon} \cap \mathrm{EmptyGraph}_{\Upsilon})) \neq \emptyset$.

Proof. The proof of this lemma can be done analogously to the one for lemma 4.27. The only difference is, that, for the parallel execution of two ASM programs, one has to build the sum of the two honest evaluation times and so for the micro times to receive the one of the resulting program. Because the Speed-Up-Theorem does not hold, one has to work with the constant factor 3. This is also the reason for the more complicated condition for f_2. $\square$

As for macroteps, the following corollary is an obvious consequence of the preceding lemma:

Corollary 4.30. *Let* $\Upsilon \in \mathrm{voc}$ *be a diagonalization-solvent, order-solvent and copy-solvent vocabulary. Let* $f_2 : \mathbb{N} \to \mathbb{N}$ *be a micro time constructible function and* $f_1 : \mathbb{N} \to \mathbb{N}$. *If* $f_2(n) \geq (|\Upsilon_{\mathrm{dyn}}| \cdot \max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g) + 1) \cdot n)^{\max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g))+1}$ *and* $f_1 = o(f_2)$ *then* $\mathrm{TIME}_{\Upsilon}^{\mathrm{micro}}(3 \cdot f_2(n)) - \mathrm{TIME}_{\Upsilon}^{\mathrm{micro}}(f_1(n)) \neq \emptyset$.

From corollaries 4.28, 4.30 and the Speed-Up-Theorem, one gets directly the following corollary:

Corollary 4.31. *Let* $\Upsilon \in \mathrm{voc}$ *be a diagonalization-solvent, order-solvent and copy-solvent vocabulary. Let* $f_2 : \mathbb{N} \to \mathbb{N}$ *be a macro time constructible function* Υ *and* $f_1 : \mathbb{N} \to \mathbb{N}$. *If* $f_2(n) \geq n^{\max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g))}$ *and* $f_1 = o(f_2)$ *then* $\mathrm{TIME}_{\Upsilon}^{\mathrm{macro}}(f_2(n)) - \mathrm{TIME}_{\Upsilon}^{\mathrm{micro}} \neq \emptyset$.

The following lemma says even more:

Lemma 4.32. *Let $\Upsilon \in \mathrm{voc}$ be a diagonalization-solvent, order-solvent and copy-solvent vocabulary. Let $f : \mathbb{N} \to \mathbb{N}$ be a macro time constructible function, If $f(n) \geq n^{\max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g))}$ then*

$$\begin{aligned} & (\mathrm{TIME}^{\mathrm{macro}}_{\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_{\Upsilon} \cap \mathrm{EmptyGraph}_{\Upsilon})) \\ - \quad & (\mathrm{TIME}^{\mathrm{micro}}_{\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_{\Upsilon} \cap \mathrm{EmptyGraph}_{\Upsilon})) \neq \emptyset \end{aligned}$$

Proof. It will be proven that for every diagonalization-solvent vocabulary, order-solvent and copy-solvent vocabulary $\Upsilon \in \mathrm{voc}$, $\mathrm{TIME}^{\mathrm{macro}}_{\Upsilon}(f(n))$ allows diagonalization over $\mathrm{TIME}^{\mathrm{micro}}_{\Upsilon}(f(n))$ concerning $\mathrm{ActiveOrder}_{\Upsilon} \cap \mathrm{EmptyGraph}_{\Upsilon}$. From the diagonalizition theorem, one can then conclude that this lemma holds. In the following, an ASM $\mathcal{A}_{\mathrm{Diag}} = (\mathrm{Diag}, (\mathrm{ActiveOrder}_{\Upsilon} \cap \mathrm{EmptyGraph}_{\Upsilon}))$ will be constructed which satisfies the conditions of definition 3.14. Diag works as follows: Every $I \in \mathrm{ActiveOrder}_{\Upsilon} \cap \mathrm{EmptyGraph}_{\Upsilon}$ is interpreted as an ASM $\mathcal{A}$, i.e., $I = I(\Upsilon, \mathcal{A})$ (it is assumed that I encodes an ASM, if not, I is rejected). Diag simulates the work of P on I. The simulation can be done within $c_P \cdot \mathrm{TIME}^{\mathrm{macro}}(P, I)$ macrosteps where c_P is a constant depending only on P. Diag accepts I iff P stops in $f_2(\mathrm{size}(I))$ macrosteps and rejects. This is achieved as follows. As in the proof of lemma 4.18, construct a disjoint structure which is totally linearly ordered by $\mathrm{SuccActive}, \mathrm{firstActive}, \mathrm{lastActive}$. The order contains $\mathrm{size}(I)$ elements. In parallel to the simulation of P, an ASM is executed on the order which satisfies the property of definition 4.17 (because f_2 is macro time constructible concerning Υ such an ASM does exist). Within one macrostep of the resulting program both programs are executed once. After $f_2(\mathrm{size}(I))$ macrosteps, Diag stops and accepts iff P has stopped and rejected.
Because of the Speed-Up-Theorem and lemma 3.2, the disjoint structure can be produced within $(\mathrm{size}(I))^{\max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g))}$ macrosteps.
Because of lemmas 3.2, 3.3 and the Speed-Up-Theorem, Diag can be constructued such that for every $J \in \mathrm{ActiveOrder}_{\Upsilon} \cap \mathrm{EmptyGraph}_{\Upsilon}$, it is true that $\mathrm{TIME}^{\mathrm{macro}}(\mathrm{Diag}, J) \leq f(\mathrm{size}(J))$.
If $\mathrm{TIME}^{\mathrm{micro}}(P, J) \leq f(\mathrm{size}(J))$ for every structure J, then the simulation halts because the number of microsteps is greater or equal to the number of macrosteps (beause per one macrostep at least one microstep is done). So, in this case, one gets that I is accepted by Diag iff I is rejected by P.
So, by use of the diagonalization theorem, this lemma can be concluded. □

The following corollary is a direct consequence of the preceding lemma:

Corollary 4.33. *Let $\Upsilon \in \mathrm{voc}$ be a diagonalization-solvent, order-solvent and copy-solvent vocabulary. Let $f : \mathbb{N} \to \mathbb{N}$ be a macro time constructible function, If $f(n) \geq n^{\max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g))}$ then $\mathrm{TIME}^{\mathrm{macro}}_{\Upsilon}(f(n)) - \mathrm{TIME}^{\mathrm{micro}}_{\Upsilon}(f(n)) \neq \emptyset$.*

4.6 Determinism And Non-Determinism

The "only" difference between deterministic and non-deterministic ASMs is the use of "guess-element" or "guess-rule".
An interesting question is, which relationships between determinism and non-determinism exist concerning time complexity. This is the subject of this section.

Lemma 4.34. *Let* $f : \mathbb{N} \to \mathbb{N}$, $i \in \{\mathrm{el}, \mathrm{rule}\}$. *Then the following inclusions hold:*

1. $\mathrm{TIME}^{\mathrm{macro}}(f(n)) \subseteq \mathrm{NTIME}_i^{\mathrm{macro}}(f(n))$
2. $\mathrm{TIME}^{\mathrm{micro}}(f(n)) \subseteq \mathrm{NTIME}_i^{\mathrm{micro}}(f(n))$

Proof. Trivial. □

The following theorem is comparable to an inclusion of complexity classes in classical complexity theory.

Theorem 4.35. *Let* $\Upsilon \in \mathrm{voc}$ *be an order-solvent and graph-solvent vocabulary and* $f : \mathbb{N} \to \mathbb{N}$.
Then

$$\begin{aligned} & \mathrm{NTIME}^{\mathrm{macro}}_{\mathrm{rule},\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon) \\ \subseteq \; & \mathrm{TIME}^{\mathrm{macro}}_\Upsilon(2^{O(f(n))}) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. Let $\Upsilon \in \mathrm{voc}$ be an order-solvent and graph-solvent vocabulary and $f : \mathbb{N} \to \mathbb{N}$.
Let $(L_1, L_2) \in \mathrm{NTIME}^{\mathrm{macro}}_{\mathrm{rule},\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon)$. So, there exists a rule-non-deterministic ASM $\mathcal{A} = (P, L_2)$ such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{NTIME}^{\mathrm{macro}}_{\mathrm{rule}}(P, J) \leq f(\mathrm{size}(J))$ for all $I \in L_2$.
Per one macrostep of P, there are at most 2 possibilities to continue per one guess-rule rule in P (apart from the infinitely many possibilities for importing an element of the reserve). Let k be the number of guess-rule rules in P. For every state of a run of P, there are at most 2^k possible successors. So, per one macrostep, at most 2^k structures can be reached from a given structure.
Let $I \in L_2$. In $f(\mathrm{size}(I))$ macrosteps, there are $(2^k)^{f(\mathrm{size}(I))} = 2^{k \cdot f(\mathrm{size}(I))}$ states reachable from I (within $f(\mathrm{size}(I))$ microsteps) by use of P.
Build up a directed graph whose nodes correspond to the states (reachable from I within $f(\mathrm{size}(I))$ macrosteps by use of P, namely the computation graph. The edges will be given by the interpretation of E.
This can be done in the following way:

First step: The input I is given. Import an element v from the reserve and execute the following updates: $\mathrm{last} := v; E(v, \mathrm{firstActive}) := \mathrm{true};$
$E(\mathrm{lastActive}, v) := \mathrm{true}.$

Second step: If I is enabling then double I (i.e., copy it). firstActive and lastActive are interpreted as the first and last element of the copy. The first and last element of the original structure are indicated by last (which is now equal to the imported element). Update the copy of I as it is indicated (choosing the first possibility) by P. If this leads to an accepting state then accept I. If this leads to an rejecting state then do the same with the next update. Else, import another element v from the reserve and execute the following updates: $E(\text{last}, v) := \text{true}; E^{-1}(v, \text{last}) := \text{true}; \text{last} := v$.
If I is disabling then reject.

step $i+1$**:** Until now, a part of the computation graph has been constructed and the last constructed vertex I_{last} has to be considered which is indicated the content of last.
If I_{last} is enabling and it has not been appeared in the already constructed part of the graph then choose the first possibility for a rule given by the program. If there is no such rule left., i.e., all of them have already been considered, then go back to the last predecessor and do the same with this state instead of I_{last}. Execute the update indicated by the chosen rule on I_{last}. If the resulting structure is accepting then accept I. If this structure is rejecting then execute the next update on I_{last}. If this structure is not accepting nor rejecting has not been reached until now then import another element v from the reserve and execute the following two updates: $E(\text{last}, v) := \text{true}; E^{-1}(v, \text{last}) := \text{true}; \text{last} := v$. If this structure has already been reached then take the corresponding element and update E by $E(\text{last}, w) := \text{true}; E^{-1}(w, \text{last}) := \text{true};$. Whether a structure has already been reached can be checked by going back the chain given by E resp. E^{-1} and comparing. Continue with the next possible update. If there is none then do the same as already described .
If there is no possibility to continue as described above then stop.

The proceeding of this deterministic algorithm is similar to a depth-first search on the computation graph starting at the input where the computation graph is constructed while executing the search.
It accepts I iff there exists an accepting state.
Because $\leq 2^{k \cdot f(\text{size}(I))}$ states are reachable from I within $f(\text{size}(I))$ macrosteps, the computation graph does contain $\leq 2^{k \cdot f(\text{size}(I))} + 1$ vertices. Every vertex of the graph has size $\leq \text{size}(I) + c_P \cdot f(\text{size}(I))$ where c_P is a constant depending only on P. To determine whether a vertex resp. a state has already been reached, try for every already constructed vertex all injective mapping and control whether it is an isomorphism. The reserve and the background part, which is not active, do not need to be considered. Therefore, only the active elements have to be considered. The whole algorithm can be executed within $2^{O(f(\text{size}(I)))} \cdot 2^{O(f(\text{size}(I)))} \cdot 2^{O(f(\text{size}(I)))} = 2^{O(f(\text{size}(I)))}$ macrosteps. □

In the special case that $\Upsilon \in \text{voc}_{\text{finite}}$, one can even prove a similar proposition for element-non-determinism.

Theorem 4.36. *Let $\Upsilon \in \text{voc}_{\text{finite}}$ be an order-solvent and graph-solvent vocabulary, $f : \mathbb{N} \to \mathbb{N}$ and $f(n) \geq n$.*
Then

$$\begin{aligned} & \text{NTIME}^{\text{macro}}_{\text{el},\Upsilon}(f(n)) \upharpoonright (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \\ \subseteq \; & \text{TIME}^{\text{macro}}_{\Upsilon}(2^{O(f(n)\cdot\log(f(n)))}) \upharpoonright (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. Let $\Upsilon \in \text{voc}_{\text{finite}}$ be an order-solvent and graph-solvent vocabulary and $f : \mathbb{N} \to \mathbb{N}$.
Let $(L_1, L_2) \in \text{NTIME}^{\text{macro}}_{\text{el},\Upsilon}(f(n)) \upharpoonright (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon)$. So, there exists a rule-non-deterministic ASM $\mathcal{A} = (P, L_2)$ such that $L_1 = \text{Acc}(\mathcal{A})$ and $\text{NTIME}^{\text{macro}}_{\text{el}}(P, J) \leq f(\text{size}(J))$ for all $I \in L_2$.
First, the number of states that can be reached from an input state I in $f(\text{size}(I))$ macrosteps of an ASM $\mathcal{A} = (P, L_2)$, such that the number of microsteps per one macrostep of P is limited by c, is computed.
Given an input structure $I \in L_2$. Define $z := \text{size}(I) + c \cdot f(\text{size}(I))$. For every structure J that appears in a run of P on I where $\text{NTIME}^{\text{macro}}_{\text{el}}(P, K) \leq f(\text{size}(K))$ (and therefore $\text{NTIME}^{\text{micro}}_{\text{el}}(P, K) \leq c \cdot f(\text{size}(K))$ for all Υ-structures K it is true that

- J contains at most z elements.
- For every function name $g \in \Upsilon$ there are at most z elements where g is not undefined (as elements of $\text{living}^{\text{arity}(g)}$) and the range contains at most z elements.

Define $y := |\Upsilon|$. So, for every $g \in \Upsilon$ there are at most $z^z = 2^{z\cdot\log(z)}$ possiblities to interprete it in a structure J with $\text{size}(J) = z$. So, there are at most $(2^{z\cdot\log(z)})^y$ structures J with $\text{size}(J) = z$ and universe $\{1, ..., z\}$. Every structure of size $\leq z$ can be embedded into a structure of size equal to z. Because Υ and therefore y is fixed, one gets $2^{O(z\cdot\log(z))}$ as result and from this also $2^{O(\text{size}(I)+f(\text{size}(I))\cdot\log(\text{size}(I)+f(\text{size}(I))))}$.
As in the proof of of lemma 4.35, one can give a deterministic algorithm which decides whether an accepting state is reachable within $f(\text{size}(I))$ macrosteps. Instead of guessing a rule, one of the finitely many living elements is guessed.
This deterministic algorithm accepts I iff there exists an accepting state which is reachable within $f(\text{size}(I))$ macrosteps. The algorithm needs to do $2^{O(\text{size}(I)+f(\text{size}(I))\cdot\log(\text{size}(I)+f(\text{size}(I))))}$ steps where every step does $2^{O(\text{size}(I)+f(\text{size}(I))\cdot\log(\text{size}(I)+f(\text{size}(I))))}$ macrosteps. So, the macro time complexity of

the corresponding ASM is

$$\leq 2^{O(\text{size}(I)+f(\text{size}(I))\cdot\log(\text{size}(I)+f(\text{size}(I))))} \cdot 2^{O(\text{size}(I)+f(\text{size}(I))\cdot\log(\text{size}(I)+f(\text{size}(I))))}$$
$$= 2^{O(\text{size}(I)+f(\text{size}(I))\cdot\log(\text{size}(I)+f(\text{size}(I))))}$$

For $f(n) \geq n$, the theorem can be concluded. □

Corollary 4.37. *Let* $\Upsilon \in \text{voc}_{\text{finite}}$ *be an order-solvent and graph-solvent vocabulary and* $f : \mathbb{N} \to \mathbb{N}$.
Then

$$\begin{aligned} & \text{NTIME}^{\text{macro}}_{\text{el},\Upsilon}(f(n)) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \\ \subseteq\ & \text{TIME}^{\text{macro}}_{\Upsilon}(2^{O((n+f(n))\cdot\log(n+f(n)))}) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. This corollary is a direct consequence of theorem 4.37 where $f(n) \geq n$ is left out. □

Theorem 4.38. *Let* $\Upsilon \in \text{voc}$ *be an order-solvent and graph-solvent vocabulary and* $f : \mathbb{N} \to \mathbb{N}$.
Then

$$\begin{aligned} & \text{NTIME}^{\text{micro}}_{\text{rule},\Upsilon}(f(n)) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \\ \subseteq\ & \text{TIME}^{\text{micro}}_{\Upsilon}(2^{O(f(n))}) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. Let $\Upsilon \in \text{voc}$ be an order-solvent and graph-solvent vocabulary and $f : \mathbb{N} \to \mathbb{N}$.
Let $(L_1, L_2) \in \text{NTIME}^{\text{micro}}_{\text{rule},\Upsilon}(f(n)) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon)$. So, there exists a rule-non-deterministic ASM $\mathcal{A} = (P, L_2)$ such that $L_1 = \text{Acc}(\mathcal{A})$ and $\text{NTIME}^{\text{micro}}_{\text{rule}}(P, J) \leq f(\text{size}(J))$ for all $I \in L_2$.
Per one microstep of P, there are at most 2 possibilities to continue (because in the considered step a guess-rule rule might be executed).
Let $I \in L_2$. Within $f(\text{size}(I))$ microsteps, $2^{f(\text{size}(I))}$ states are reachable from I by use of P.
The rest of the proof can be done analogously to the proof of theorem 4.35. □

Again, in the special case Υ_{finite}, one can prove a similar proposition

Theorem 4.39. *Let* $\Upsilon \in \text{voc}_{\text{finite}}$ *be an order-solvent and graph-solvent vocabulary and* $f : \mathbb{N} \to \mathbb{N}$.
Then

$$\begin{aligned} & \text{NTIME}^{\text{micro}}_{\text{el},\Upsilon}(f(n)) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \\ \subseteq\ & \text{TIME}^{\text{micro}}_{\Upsilon}(2^{O((n+f(n))^2\cdot\log(n+f(n)))}) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. Let $\Upsilon \in \text{voc}_{\text{finite}}$ be an order-solvent and graph-solvent vocabulary and f be a micro time constructible function.
Let $(L_1, L_2) \in \text{NTIME}^{\text{micro}}_{\text{rule},\Upsilon}(f(n)) \upharpoonright (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon)$. So, there exists a rule-non-deterministic ASM $\mathcal{A} = (P, L_2)$ such that $L_1 = \text{Acc}(\mathcal{A})$ and $\text{NTIME}^{\text{micro}}_{\text{rule}}(P, J) \le f(\text{size}(J))$ for all $I \in L_2$.
First, the number of states that can be reached from a state $I \in L_2$ in $f(\text{size}(I))$ microsteps of an ASM $\mathcal{A} = (P, L_2)$ is computed.
Given $I \in L_2$. Define $z := \text{size}(I) + f(\text{size}(I))$. For every structure J that appears in a run of P on I where $\text{NTIME}^{\text{micro}}_{\text{el,finite}}(P, K) \le f(\text{size}(K))$ for all Υ-structures K it is true that

- J contains at most active z elements.
- For every function name $g \in \Upsilon$ there are at most z elements (as elements of $\text{living}^{\text{arity}(g)}$) and the range contains at most z elements.

Define $y := |\Upsilon|$. So, for every $g \in \Upsilon$ there are at most $z^z = 2^{z\cdot\log(z)}$ possibilities to interprete it in a structure J with $\text{size}(J) = z$. So, there are at most $(2^{z\cdot\log(z)})^y = 2^{y\cdot z\cdot\log(z)}$ structures J with $\text{size}(J) = z$. Υ and therefore y is fixed by the program P and, w.l.o.g., one can assume that $y \le \text{size}(I)$ So, one gets $2^{O(z^2\cdot\log(z))}$ as result. The rest of the proof can be done analogously to the proof of the preceding lemma for macrosteps, i.e., the given program must be adopted and the arguments for the computational complexity can be done analogously to those for macrosteps. $\square$

Corollary 4.40. *Let $\Upsilon \in \text{voc}_{\text{finite}}$ be an order-solvent and graph-solvent vocabulary and $f : \mathbb{N} \to \mathbb{N}$.*
Then

$$\begin{aligned} & \text{NTIME}^{\text{micro}}_{\text{el},\Upsilon}(f(n)) \upharpoonright (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \\ \subseteq\ & \text{TIME}^{\text{micro}}_{\Upsilon}(2^{O(|\Upsilon|\cdot(n+f(n))\cdot\log(n+f(n)))}) \upharpoonright (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. The proof of theorem 4.39 already contains the proof of this corollary. $\square$

4.7 The Classes P And NP

In this chapter, NP-complete and P-complete sets of structures will be considered. There are four complexity classes $\text{NP}^{\text{macro}}_{\text{el}}$, $\text{NP}^{\text{micro}}_{\text{el}}$, $\text{NP}^{\text{macro}}_{\text{rule}}$, $\text{NP}^{\text{micro}}_{\text{rule}}$.

Definition 4.41. Let pol be the set of all polynomials. For $i \in \{\text{el}, \text{rule}\}$, $\Upsilon \in$

voc define:

$$\begin{aligned}
\mathrm{P}^{\mathrm{macro}}_{\Upsilon} &:= \bigcup_{p\in\mathrm{pol}} \mathrm{TIME}^{\mathrm{macro}}_{i,\Upsilon}(p(n))\\
\mathrm{P}^{\mathrm{micro}}_{\Upsilon} &:= \bigcup_{p\in\mathrm{pol}} \mathrm{TIME}^{\mathrm{micro}}_{i,\Upsilon}(p(n))\\
\mathrm{P}^{\mathrm{macro}} &:= \bigcup_{p\in\mathrm{pol}} \mathrm{TIME}^{\mathrm{macro}}_{i}(p(n))\\
\mathrm{P}^{\mathrm{micro}} &:= \bigcup_{p\in\mathrm{pol}} \mathrm{TIME}^{\mathrm{micro}}_{i}(p(n))\\
\mathrm{NP}^{\mathrm{macro}}_{i,\Upsilon} &:= \bigcup_{p\in\mathrm{pol}} \mathrm{NTIME}^{\mathrm{macro}}_{i,\Upsilon}(p(n))\\
\mathrm{NP}^{\mathrm{micro}}_{i,\Upsilon} &:= \bigcup_{p\in\mathrm{pol}} \mathrm{NTIME}^{\mathrm{micro}}_{i,\Upsilon}(p(n))\\
\mathrm{NP}^{\mathrm{macro}}_{i} &:= \bigcup_{p\in\mathrm{pol}} \mathrm{NTIME}^{\mathrm{macro}}_{i}(p(n))\\
\mathrm{NP}^{\mathrm{micro}}_{i} &:= \bigcup_{p\in\mathrm{pol}} \mathrm{NTIME}^{\mathrm{micro}}_{i}(p(n))
\end{aligned}$$

According to lemma 2.5, lemma 2.6 and lemma 2.7, every ASM-program is equivalent to an ASM-program of the respective normal form. From the proof one can see the following: If there is a program which works within polynomial time (compared to the size of the input structure) then there is an equivalent ASM-program in the respective normal form which also works in within polynomial time. This is indicated by the construction of the normal form from the given program. This construction preserves the polynomial time limitation.
To be more general, one say even say that for all three normal forms, the following holds:
The number of macrosteps done by the original program during a run is equal to the number of macrosteps done by the transformed program in normal form. This is indicated by the equivalence of the programs.
Furthermore, for every program P there exists a constant $c_P \geq 1$ such that the number of microsteps done by the original program during a run is equal to the number of microsteps done transformed program in normal form multiplied by c_P.

Definition 4.42. Let $c \in \mathbb{N}$. An ASM-program P is called *c-micro-limited* if per one macrostep of P, P does at most c microsteps.

Note: The number of microsteps done at one macrostep is limited by the number of nodes in the concrete parse tree. There is no upper bound for the constant c in general.

Theorem 4.43. *Let $i \in \{\mathrm{el}, \mathrm{rule}\}$. Then the following equations hold:*

1. $\mathrm{P}_{\Upsilon}^{\mathrm{micro}} = \mathrm{P}_{\Upsilon}^{\mathrm{macro}}$
2. $\mathrm{P}^{\mathrm{micro}} = \mathrm{P}^{\mathrm{macro}}$
3. $\mathrm{NP}_{i,\Upsilon}^{\mathrm{micro}} = \mathrm{NP}_{i,\Upsilon}^{\mathrm{macro}}$
4. $\mathrm{NP}_{i}^{\mathrm{micro}} = \mathrm{NP}_{i}^{\mathrm{macro}}$

Proof. The proof of 1. will be given. The proof of 3. can be done analogously to the one of 1.. 2. is a consequence of 1., and 4. is a consequence of 3..
$\mathrm{P}_{\Upsilon}^{\mathrm{micro}} \subseteq \mathrm{P}_{\Upsilon}^{\mathrm{macro}}$ holds because $\mathrm{TIME}_{\Upsilon}^{\mathrm{micro}}(f(n)) \subseteq \mathrm{TIME}_{\Upsilon}^{\mathrm{macro}}(f(n))$ for any $f : \mathbb{N} \to \mathbb{N}$ (this has already been proven).
In the rest of this proof, $\mathrm{P}_{\Upsilon}^{\mathrm{macro}} \subseteq \mathrm{P}_{\Upsilon}^{\mathrm{micro}}$ will be shown. Let $(L_1, L_2) \in \mathrm{P}_{\Upsilon}^{\mathrm{macro}}$ be a pair of sets of Υ-structures of finite size, $L_1 \subseteq L_2$. So, there exist a polynomial p and a (deterministic) ASM $\mathcal{A} = (Q, L_2)$ such that Q does at most $p(\mathrm{size}(I))$ macrosteps on any structure $I \in L_2$. There exists a constant c such that Q is c-micro-limited. So, Q does at most $c \cdot p(\mathrm{size}(I))$ microsteps on any sructure $I \in L_2$. Because $c \cdot p(n)$ is also a polynomial, $(L_1, L_2) \in \mathrm{P}_{\Upsilon}^{\mathrm{micro}}$. □

As a consequence of theorem 4.43, the following definition makes sense:

Definition 4.44. Let $i \in \{\mathrm{el}, \mathrm{rule}\}$.

1. $\mathrm{P}_{\Upsilon} := \mathrm{P}_{\Upsilon}^{\mathrm{macro}} (= \mathrm{P}_{\Upsilon}^{\mathrm{micro}})$
2. $\mathrm{P} := \mathrm{P}^{\mathrm{macro}} (= \mathrm{P}^{\mathrm{micro}})$
3. $\mathrm{NP}_{i,\Upsilon} := \mathrm{NP}_{i,\Upsilon}^{\mathrm{macro}} (= \mathrm{NP}_{i,\Upsilon}^{\mathrm{micro}})$
4. $\mathrm{NP}_{i} := \mathrm{NP}_{i}^{\mathrm{macro}} (= \mathrm{NP}_{i}^{\mathrm{micro}})$

Lemma 4.45. *Let $\Upsilon \in \mathrm{voc}$. Then the following inclusions hold:*

1. $\mathrm{NP}_{\mathrm{rule},\Upsilon} \subseteq \mathrm{NP}_{\mathrm{el},\Upsilon}$
2. $\mathrm{NP}_{\mathrm{rule}} \subseteq \mathrm{NP}_{\mathrm{el}}$

Proof. Because the guess-rule rule can be simulated by the guess-element rule as shown in section 2.4 with only constant overhead per turn (macrostep) per rule one can conclude this theorem. □

Because of theorem 4.43, $\mathrm{NP}_{i}^{\mathrm{macro}}$-completeness and $\mathrm{NP}_{i}^{\mathrm{micro}}$-completeness are equivalent for $i \in \{\mathrm{el}, \mathrm{rule}\}$ (which will be called NP_i-completeness in the following). So, there is no need to make a difference between macrosteps and microsteps in this section. So, for proofs, always choose the more proper formulation.
Furthermore, in the following the term "polynomial time" will be used without mentioning the time measure (macrosteps resp. microsteps). This makes sense because of the previous.

4.8 A Logic Captured By P

In the following, a logic is considered as a set of formulae with a given semantics. Formulae are considered as trees.
The following definitions are similar to the definitions of number terms, $\mathrm{FO}_{\mathbb{R}}$ and $\mathrm{FO}_{\mathbb{R}}$-reduction in [7].
Fix a countable set of variables $V = \{x_0, x_1, ...\}$. These variables range only over the active part of a structure or Boole (and not over the background).
Fix another countable set of variables $R = \{r_0, r_1, ...\}$. These variables only over the reserve of a structure.
Atomic formulae are equalities $t = t'$ where t and t' are terms (and terms are defined as usual).

Definition 4.46. Define $\mathrm{FO}_{\mathrm{static}}$ as follows:
The set of formulae in $\mathrm{FO}_{\mathrm{static}}$ is the smallest set that contains all atomic formulae and is closed under Boolean connectives, quantification $(\exists x)\psi$ and $(\forall x)\psi$, where $x \in V$ (so, these bound variables range only over the active part of a structure in which it is interpreted) and quantification $(\exists r)\psi$ and $(\forall r)\psi$, where $r \in R$ (so, these variables range only over the reserve).

It is important that no formula $(\exists y)\psi$ or $(\forall y)\psi$ is built where y ranges over the background and non-active part (without the obligatory background part).
In the following, an $\mathrm{FO}_{\mathrm{static}}$-formula is considered as a tree.

Lemma 4.47. *Let φ be an $\mathrm{FO}_{\mathrm{static}}$-formula over an order-solvent vocabulary $\Upsilon \in$ voc and $L_2 \in$ ActiveOrder.*
Then $\{I : I \in L_2$ and $I \models \varphi\}, L_2) \in$ P.

Proof. W.l.o.g., it can be assumed that φ is given in prenex normal form (i.e., all quantifiers are in front followed by a quantifier free part ψ). Let I be an input structure over Υ such that the active part of the universe of I is linearly, totally ordered by Succ with min as least element. So, every active element is directly accessible as $\mathrm{Succ}^n(\min)$ for some $n \in \mathbb{N}$.
Because every quantifiers ranges only over the active part or only over the reserve (and they are fixed and therefore their number), for a given input structure there are only polynomially many different possible assignments for the variables in ψ (to be more precise, there are infinitely many but the reserve is a naked set with only equality defined on it and therefore, only the cases concerning differently stuctured concerning equality have to be considered; the number of these cases does only depend on the formula). Evaluate ψ for all of them. Every term occuring in ψ has only constant length and can therefore be evaluated in polynomial time. So, the truth value of ψ on any of the polynomially many (compared to the size of the formula) assignments can be computed in polynomial time. So, one gets the same result for φ. ☐

4.9 A Logical Description Of $\mathbf{NP}_{\mathrm{el}}$

In this section, a theorem with a proposition similar to the one of Fagin's Theorem will be shown.
The second-order logic stat$-\Sigma_1^1$ is obtained from $\mathrm{FO}_{\mathrm{static}}$ by adding the possibility to quantify existentially over function names f such that the active domain of f^I contains only finitely many elements of the reserve, contains only finitely many element of Background(I) and arbitrary many elements of Active(I) (which is finite and therefore, it only contain finitely many elements of Active(I)).

Definition 4.48. φ is an *existential second-order sentence* (of vocabulary Υ) if $\varphi = \exists f_1 ... \exists f_n \psi$ where ψ is a first-order sentence in $\mathrm{FO}_{\mathrm{static}}$ of vocabulary $\Upsilon \cup \{f_1, ..., f_n\}$ and f_1, ..., f_n are not static functions and have the property described above.

Theorem 4.49. *Let $\Upsilon \in \mathrm{voc}$ be an order-solvent vocabulary.*

$$\mathrm{stat}{-}\Sigma_1^1(\Upsilon) = \mathrm{NP}_{\mathrm{el}} \restriction \mathrm{ActiveOrder}_\Upsilon$$

More precisely: Let $L_2 \in \mathrm{ActiveOrder}_\Upsilon, L_1 \subseteq L_2$ then the following holds:
$(L_1, L_2) \in \mathrm{NP}_{\mathrm{el},\Upsilon}$
iff
there exists an existential second-order sentence ψ over Υ such that
$L_1 = \{I : I \in L_2$ *and* $I \models \psi\}$

Proof. In this proof, the characterization of $\mathrm{NP}_{\mathrm{el}}$ via macrosteps will be used.
First, it will be shown that "stat$-\Sigma_1^1 \subseteq \mathrm{NP}_{\mathrm{el}}$ ". This is true, because, given a stat$-\Sigma_1^1$-sentence ψ (a stat$-\Sigma_1^1$-formula is considered as a tree), for a structure I (over a large enough vocabulary) in element-non-deterministic polynomial (concerning size(I)) time it can be checked whether $I \models \psi$ (the disjoint union of I and the tree representing φ gives a structure). An algorithm, which certifies this, is described in the following. So, given a sentence $\psi = \exists f_1 ... \exists f_k \varphi \in \mathrm{stat}{-}\Sigma_1^1$ where φ is an $\mathrm{FO}_{\mathrm{static}}$-sentence, guess $f_1, ..., f_k$ (with the guess-element rule, for f_i with arity r_i, one can guess for every r_i-tuple of elements from Active(I) a living element or undef). Then verify, that $(I, T_1, ..., T_k) \models \varphi$ (where $(I, f_1, ..., f_k)$ is the structure I together with the guessed functions). Only polynomially many guesses are necessary (in the size of the input structure).
Now, it will be proven that "$\mathrm{NP}_{\mathrm{el}} \subseteq \mathrm{stat}{-}\Sigma_1^1$", i.e., for every $(L_1, L_2) \in \mathrm{NP}_{\mathrm{el}}$ (and therefore, $(L_1, L2) \in \mathrm{NP}_{\mathrm{el},\Upsilon}$ for an $\Upsilon \in \mathrm{voc}$) there exists a sentence $\psi \in \mathrm{stat}{-}\Sigma_1^1$ such that $L_1 = \{I : I \in L_2$ and $I \models \psi\}$.
W.l.o.g., one can assume that the active part of all Υ-structures is ordered (and has a successor function) by the order $\succ$ indicated by the interpretations of firstActive, lastActive, SuccActive. Else, one has to extend the formula ψ, which will be constructed in the following, to a formula $\psi' = \exists \succ (\psi \wedge \mathrm{order}(\succ, \mathrm{firstActive}, \mathrm{lastActive}, \mathrm{SuccActive}))$ where order is an $\mathrm{FO}_{\mathrm{static}}$-formula, that is

the conjunction of the axioms for the total order $\succ$ indicated by the interpretations of $\mathrm{firstActive}, \mathrm{lastActive}, \mathrm{SuccActive}$. ψ'' can directly be transformed into an equivalent $\Sigma_1^1(\Upsilon)$-sentence.
There exists $r \in \mathbb{N}$ such that $\mathrm{size}(I) \leq |\mathrm{Active}(I)|^r$ (r depends on Υ). Because $\mathcal{K} \in \mathrm{NP}_{\mathrm{el},\Upsilon}$, there exist a element-non-determinstic ASM $\mathcal{A} = (Q, L_2)$ and $s \in \mathbb{N}$ such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{NTIME}_{\mathrm{el}}^{\mathrm{macro}}(Q, I) \leq (\mathrm{size}(I))^s$. So, there exists $k \in \mathbb{N}$ such that $\mathrm{NTIME}_{\mathrm{el}}^{\mathrm{macro}}(Q, I) \leq (|\mathrm{Active}(I)|)^k$ ($k = s + r$) for every $I \in L_2$. So, time can be described by elements of $(\mathrm{Active}(I))^k$ (the order on $(\mathrm{Active}(I))^k$ is the lexicographical order resulting from $\succ$, this order will be called $>$ in the following. In the following, the t-th element of this lexikographical order will be identified with t. Let $t_{\max}$ be its maximal element. $t+1$ and $t-1$ can be defined as usual.
The stat-Σ_1^1-formula ψ, that will be constructed, roughly says, that there exists an accepting run of Q on input I (so, $I \in L_1$).
W.l.o.g., one can assume that Q has the following normal form (according to lemma 2.6 and the remark in section 4.7):

```
import v_1
  .
  .
  .
import v_l
  guess-element w_1 :  g_1(w_1)
    .
    .
    .
  guess-element w_m :  g_m(w_m)
    block
      if c_1 then R_1 endif
        .
        .
        .
      if c_n then R_n endif
    endblock
  endguess-element
    .
    .
    .
  endguess-element
```

```
endimport
    .
    .
    .
endimport
```

where R_i, $i \in \{1, ..., m\}$ are updates.
ψ has the following form:

$$\psi = \exists \mathrm{IMP}_1 ... \exists \mathrm{IMP}_l \exists \mathrm{GUESS}_1 ... \exists \mathrm{GUESS}_m \exists F_1 ... \exists F_n \forall t \varphi$$

where $\Upsilon = \{f_1, ..., f_n\}$, l is the number of import rules in the ASM-program, m is the number of guess-element rules in the ASM-program, $\forall t$ is the short version of $\forall x_1 ... \forall x_r$ ($x_1, ..., x_r \in V$ and therefore, these variables range only over the active part of the structure) and φ is an $\mathrm{FO}_{\mathrm{static}}$-formula looking as follows:

$$\varphi = \mathrm{START} \wedge \mathrm{COMPUTE} \wedge \mathrm{ACCEPT}$$

$\mathrm{IMP}_i, i \in \{1, ..., l\}$, has arity k. $\mathrm{IMP}_i(t) = x$ (to be more exact, instead of t one should write the k-tuple standing for time t) iff x is imported by the i-th import rule at time t and x is a reserve element at time t. The condition that x is an element from the reserve will be ensured by a formula described later.
$\mathrm{GUESS}_i, i \in \{1, ..., m\}$, has arity k. $\mathrm{GUESS}_i(t) = x$ (to be more exact, instead of t one should write the k-tuple standing for time t) iff x is guessed by the i-th guess-element rule at time t and x is a living element at time t.
$F, f \in \Upsilon - \Upsilon_{\mathrm{stat}}$ (so, all non-static function names are considered here), has arity $\mathrm{arity}(f) + k$ and $F(x_1, ..., x_{\mathrm{arity}(f)}, t) = y$ iff $f(x_1, ..., x_{\mathrm{arity}(f)}) = y$ at time t.
Now, the components of φ will be described:

START This formula "translates" the input structure into terms of F, $f \in \Upsilon - \Upsilon_{\mathrm{stat}}$. More precisely, START has the following appearance:

$$\Big(\bigwedge_{f \in \Upsilon - \Upsilon_{\mathrm{stat}}} (\forall x_1 ... \forall x_{\mathrm{arity}(f)} (F(x_1, ..., x_{\mathrm{arity}(f)}, 0) = f(x_1, ..., x_{\mathrm{arity}(f)}))))$$

COMPUTE COMPUTE will be the universal quantification over all $0 \leq t \leq t_{\max}$ where $t_{\max}$ is not fixed a priori because the size of the active part of the input is not fixed a priori, but one knows that it is $\leq |\mathrm{Active}(I)|^k$. So, it has the form $\forall t \Phi$ (which more precisely is $\forall x_1 ... \forall x_k \Phi$.
First, an update $q := p$ where p and q are terms is translated into a formula because this is the basis. Then, this will be expanded to an if-clause which will afterwards be expanded to a block of if-clauses. It is important that no unbound variable does occur in an update (because the considered updates only occur in programs which do not contain any unbound variables). The update is treated as if it does not contain any free variables (because if

the update is embedded into a program, then the only variables occuring are bound by import or guess such that they can be substituted by the corresponding elements of the universe). The component contributed by this update is $\mathrm{Eval}(q, t+1) = \mathrm{Eval}(p, t)$ where $\mathrm{Eval}(s, t)$ for a term s and time t will defined by induction on the structure of the term s as follows:

- if $s = f(y_1, ..., y_{\mathrm{arity}(f)})$ and f is not static then $\mathrm{Eval}(s, t) = F(\mathrm{Eval}(y_1, t), ..., \mathrm{Eval}(y_{\mathrm{arity}(f)}, t))$.
- if $s = f(y_1, ..., y_{\mathrm{arity}(f)})$ and f is static then $\mathrm{Eval}(s, t) = f(\mathrm{Eval}(y_1, t), ..., \mathrm{Eval}(y_{\mathrm{arity}(f)}, t))$.
- if s is a variable or a static, nullary function name then $\mathrm{Eval}(s, t) = s$.
- if s is the i-th element imported (at time t), then $\mathrm{Eval}(s, t) = \mathrm{IMP}_i(t)$.
- if s is the i-th element guessed (at time t), then $\mathrm{Eval}(s, t) = \mathrm{GUESS}_i(t)$.

Now, the case of an if-clause of the form `if` c `then` R `endif` where R is an update is considered. This case differs only slightly from the preceding one. Let $E(R, t)$ be the formula corresponding to R at time t and gained as described above. c will also be translated into the notation with the functions $F, f \in \Upsilon$ introduced as above. Call this translation (i.e., the formula corresponding to c at time t) is $E(c, t)$. The translation of the if-clause at time t will be $E(c, t) \to E(R, t)$. Now, the translation of c is described. Because c can only be a composition of equations of the type $r = s$ (where r and s are terms over the vocabulary Υ) by $\wedge$, $\vee$ and $\neg$, it is rather simple to transform c at time t into $E(c, t)$. $E(c, t)$ results from c by replacing every equation $r = s$, occuring in c by $\mathrm{Eval}(r, t) = \mathrm{Eval}(s, t)$. (These expressions are composed just like the corresponding equations to obtain $E(c, t)$).
Now, a block of if-clauses is considered.
It is not enough to build the conjunction of the formulae given by the if-clauses as described above. Any arbitrary change of the conents of locations without any indication of the program has to be avoided. I.e., if a location is not concerned by any update then its content does not change. If one only takes the sequence of the formulae gained as described above then such an arbitrary change of the content might be possible.
Therefore, all 2^l possible combinations of truth values of the guards of the if-clauses have to be considered. For every subset M of $\{1, ..., n\}$ (so, $M \in \mathcal{P}(\{1, ..., n\})$) define a formula $\psi_{M,t}$. It has the form

$$\psi_{M,t} = (\bigwedge_{i \in M} c_i \wedge \bigwedge_{j \notin M} \neg c_j) \to \varphi_{M,t}$$

$\psi_{M,t}$ describes the updates in the considered case. I.e., it gives the values in the next state. Those locations, which are concerned by the updates (this

is the case if the respective guard is satisfied), get the contents indicated by the updates. The others do not change. The updates can be grouped by the function names given in the updated location. Let $M_f := \{f(t_1^1, ..., t_r^1) := s_1, ..., f(t_1^k, ..., t_r^k) := s_k\}$ be the set of all updates which update a location with function name f and belong to the i-th if-clause where $i \in M$.
So, $\varphi_{M,t} = \bigwedge_{f \in \Upsilon - \Upsilon_{\text{stat}}} \varphi_{M,f,t}$. In $\varphi_{M,f,t}$ only those updates are considered which deal with f. Now, $\psi_{M,f,t}$ will be given:

$$
\begin{aligned}
&\varphi_{M,f,t} = \\
&\qquad \bigwedge_{f(t_1,...,t_r):=s \in M_f} \text{Eval}(f(t_1, ..., t_r), t+1) = \text{Eval}(s, t) \\
&\wedge \bigwedge_{N_1,N_2 \subseteq \{1,...,r\}, N_1 \cap N_2 = \emptyset} \ \bigwedge_{u_1,...,u_{|N_1|} \in \{1,...,l\}} \ \bigwedge_{v_1,...,v_{|N_2|} \in \{1,...,m\}} \forall t_1^{N_1} ... \forall t_{|N_1|}^{N_1} \forall t_1^{N_2} ... \forall t_{|N_2|}^{N_2} \\
&\qquad ((\bigwedge_{1 \leq i \leq |N_1|} t_i^{N_1} \leq t \wedge \bigwedge_{1 \leq i \leq |N_2|} t_i^{N_2} \leq t) \rightarrow \chi_{f,M,N,u_1,...,u_{|N_1|},v_1,...,v_{|N_2|},t_1^{N_1},...,t_{|N_1|}^{N_1},t_1^{N_2},...,t_{|N_2|}^{N_2}})
\end{aligned}
$$

where $N_1 = \{m_1, ..., m_{|M|}\}$, $m_1 < ... < m_{|N_1|}$, $N_2 = \{n_1, ..., n_{|N_2|}\}$, $n_1 < ... < n_{|N_2|}$, $N_1 \cup N_2 = \{w_1, ..., w_z\}$, $w_1 < ... < w_z$, and

$$
\begin{aligned}
&\chi_{f,N_1,N_2,u_1,...,u_{|N_1|},v_1,...,v_{|N_2|},t_1^{N_1},...,t_{|N_1|}^{N_1},t_1^{N_2},...,t_{|N_2|}^{N_2}} = \\
&\quad \forall x_1 ... \quad \forall x_{w_1-1} \forall x_{w_1+1} ... \forall x_{w_z-1} \forall x_{w_z+1} ... \forall x_r \\
&\quad \bigwedge_{j \in \{1,...,k\}} (\neg(\bigwedge_{i \in \{1,...,r\} - N_1} x_i = \text{Eval}(t_i^j, t) \wedge \bigwedge_{a \in N_1} \text{Eval}(t_a^j, t) = \text{IMP}_{u_a}(t_a^{N_1}))) \\
&\quad \wedge \quad \bigwedge_{j \in \{1,...,k\}} (\neg(\bigwedge_{i \in \{1,...,r\} - N_2} x_i = \text{Eval}(t_i^j, t) \wedge \bigwedge_{a \in N_2} \text{Eval}(t_a^j, t) = \text{GUESS}_{v_a}(t_a^{N_2}))) \\
&\quad \rightarrow \quad ((F(x_1, ..., x_r, t+1) = F(x_1, ..., , x_r, t)) \\
&\qquad [x_{m_1}/\text{IMP}_{u_1}(t_1^{N_1}), ..., x_{m_{|N_1|}}/\text{IMP}_{u_{|N_1|}}(t_{|N_1|}^{N_1}), x_{n_1}/\text{GUESS}_{v_1}(t_1^{N_2}), \\
&\qquad ..., x_{n_{|N_1|}}/\text{GUESS}_{v_{|N_2|}}(t_{|N_1|}^{N_1})]))
\end{aligned}
$$

By use of the formulae $\chi_{f,N_1,N_2,u_1,...,u_{|N_1|},v_1,...,v_{|N_2|},t_1^{N_1},...,t_{|N_1|}^{N_1},t_1^{N_2},...,t_{|N_2|}^{N_2}}$, an arbitrary change in the locations is avoided. I.e., if no update in the program indicates a change of a location then it is not changed. Because only those elements can be quantified which are active in the input structure, the imported and guessed elements have to be considered separately. N_1 gives the positions for the imported elements, N those for the guessed elements. u_i resp. v_i gives the number of the introducing rule and $t_i^{N_1}$ resp. $t_i^{N_2}$ gives the time where the element has been imported resp. guessed.

The other elements of the background which become active during the computation are already handled by use of Eval because they can only be introduced by application of functions.
Now, the import-rules are considered. $\mathrm{IMP}_i(t)$ represents the i-th element imported at time t. The following component has to be introduced to make sure that the imported elements to make sure that they are all different that they belong to the reserve.

$$\begin{aligned} &\forall t_1 \forall t_2 \bigwedge_{i,j\in\{1,\ldots,l\},i\neq j} \mathrm{IMP}_i(t_1) \neq \mathrm{IMP}_j(t_2) \\ \wedge &\bigwedge_{i\in\{1,\ldots,l\}} \exists r \mathrm{IMP}_i(t) = r \end{aligned}$$

With the first conjunct (first line) it is ensured that two elements imported at the same time by different rules are not equal. The conjunct demands even more but this is no restriction because the reserve is assumed to be infinite. Together with the first conjunct, the second conjunct (second line) ensures that the imported every imported element is from the reserve (at the considered time). This does also demand more but, because of the same reason as before, this no restriction.
Now, the i-th guess-element rule is considered. It has the form (the components must be substituted corresponding to the given program, here only a guess-element rule is considered):

> `guess-element` v_i : $g_i(v_i)$
> $\quad R(v_i)$
> `endguess-element`

$g_i(v_i)$ can be translated just like the guard of an if-clause (see above). So, add the component $\forall t' E(g_i[v/\mathrm{GUESS}_i(t')], t')$.
Furthermore, it has to be ensured that the guessed element does not come from the reserve. Here it is not right to exclude that it comes from the reserve of the input structure because such an element could have become active until the considered time by importing and updating. So, one has also to consider the imported. For every $i \in \{1, ..., m\}$ (so, for every guess-element-rule), add the following component:

$$
\begin{array}{ll}
& ((\forall r \mathrm{GUESS}_i(t) \neq r) \wedge (\forall x \mathrm{GUESS}_i(t) \neq x) \\
\wedge & (\forall t'(t' < t \to \bigwedge_{j\in\{1,\dots,m\}} \mathrm{GUESS}_i(t) \neq \mathrm{GUESS}_j(t'))) \\
\vee & \bigvee_{N_1,N_2\subseteq\{1,\dots,r\},N_1\cap N_2=\emptyset} \bigvee_{j\in\{1,\dots,r\}-(N_1\cup N_2)} \bigvee_{u_1,\dots,u_{|N_1|}\in\{1,\dots,l\}} \bigvee_{v_1,\dots,v_{|N_2|}\in\{1,\dots,m\}} \\
& \exists t_1^{N_1} \dots \exists t_{|N_1|}^{N_1} \exists t_1^{N_2} \dots \forall t_{|N_2|}^{N_2} \\
& (\bigwedge_{1\le i\le |N_1|} t_i^{N_1} \le t \wedge \bigwedge_{1\le i\le |N_2|} t_i^{N_2} \le t \wedge \Psi_{f,j,N_1,N_2,u_1,\dots,u_{|N_1|},v_1,\dots,v_{|N_2|},t_1^{N_1},\dots,t_{|N_1|}^{N_1},t_1^{N_2},\dots,t_{|N_2|}^{N_2}})
\end{array}
$$

where $N_1 = \{m_1, ..., m_{|M|}\}$, $m_1 < ... < m_{|N_1|}$, $N_2 = \{n_1, ..., n_{|N_2|}\}$, $n_1 < ... < n_{|N_2|}$, $N_1 \cup N_2 = \{w_1, ..., w_z\}$, $w_1 < ... < w_z$, and The first disjunct says that the guessed element is not from the original reserve or the original active part and that it has not been guessed until the considered time. Therefore, the guessed element has originally been in the background and non-active part and it has not been guessed until time t.
By use of the formulae $\Psi_{f,j,N_1,N_2,u_1,\dots,u_{|N_1|},v_1,\dots,v_{|N_2|},t_1^{N_1},\dots,t_{|N_1|}^{M},t_1^{N_2},\dots,t_{|N_2|}^{N_2}}$, it is expressed that at time t, the i-th guessed element is active. The construction of this formula is similar to the construction
of the formula $\chi_{f,N_1,N_2,u_1,\dots,u_{|N_1|},v_1,\dots,v_{|N_2|},t_1^{N_1},\dots,t_{|N_1|}^{M},t_1^{N_2},\dots,t_{|N_2|}^{N_2}}$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\Psi_{f,j,N_1,N_2,u_1,\dots,u_{|N_1|},v_1,\dots,v_{|N_2|},t_1^{N_1},\dots,t_{|N_1|}^{N_1},t_1^{N_2},\dots,t_{|N_2|}^{N_2}} =} \\
\exists x_1 \dots & \exists x_{w_1-1} \exists x_{w_1+1} \dots \exists x_{w_z-1} \exists x_{w_z+1} \dots \exists x_r ((F(x_1, \dots, , x_r, t) \neq \mathrm{undef}) \\
& [x_{m_1}/\mathrm{IMP}_{u_1}(t_1^{N_1}), \dots, x_{m_{|N_1|}}/\mathrm{IMP}_{u_{|N_1|}}(t_{|N_1|}^{N_1}), x_{n_1}/\mathrm{GUESS}_{v_1}(t_1^{N_2}), \\
& \dots, x_{n_{|N_1|}}/\mathrm{GUESS}_{v_{|N_2|}}(t_{|N_1|}^{N_1}), x_j/\mathrm{GUESS}_i(t)]))
\end{array}
$$

For the translation of the rest of the program at time t, substitute v_i by $\mathrm{GUESS}_i(t)$, so use $E(R[v_i/\mathrm{GUESS}_i(t)], t))$.
So, COMPUTE $= \forall t R$ where R is the formula constructed above.

ACCEPT This formula describes the condition for accepting an input structure. Let MODE be the quantified function corresponding to Mode and OUTPUT the one corresponding to Output

$$\mathrm{ACCEPT} = \exists t'(t' \le t_{\max} \wedge \mathrm{MODE}(t') = \mathrm{false} \wedge \mathrm{OUTPUT}(t') = \mathrm{true})$$

Following the construction, one gets for $I \in L_2$ that:
$I \in L_1$

iff there exists an accepting run of Q on I
iff $I \models \psi$ $\square$

As one can see from the proof of the previous theorem, $\Sigma_1^1(\Upsilon)$ (where $\Upsilon \in voc$ is an arbitrary vocabulary) captures $\mathrm{NP}_{\mathrm{el},\Upsilon}$ because one can existentially quantify a successor function and then construct the formula as described above. Therefore, Σ_1^1 captures $\mathrm{NP}_{\mathrm{el}}$.

4.10 Completeness

In the following, a special kind of reduction will be introduced: $\mathrm{FO}_{\mathrm{static}}$-reduction. First, the notion of a first-order interpretation has to be defined.

Definition 4.50. Let $\Upsilon, \Upsilon' \in$ voc. Let $k \in \mathbb{N}$ be a constant. A *first-order interpretation* (of width k) of Υ-structures in Υ'-structures is given by a set Φ of $\mathrm{FO}_{\mathrm{static}}$-formulae and terms such that:
For every r-ary function name $f \in \Upsilon$, Φ contains a formula φ_f of vocabulary Υ' with free variables $t_{i,j}$ where $i \in \{1, ..., r+1\}$, $j \in \{1, ..., k\}$ such that for every Υ'-structure I', φ_f defines a function on $\Phi I'$, where $\Phi I'$ is defined as follows:
The universe of $\Phi I'$ is universe$(I')^k$.
Let $f \in \Upsilon$ an r-ary function name. Then
$f^{\Phi I'}(t^1, ..., t^r) = t^{r+1}$ iff I' is a model for $\varphi_f(t_1^1, ..., t_1^k, ..., t_r^1, ..., t_r^k, t_{r+1}^1, ..., t_{r+1}^k)$

Definition 4.51. Let K be a set of Υ-structures and let K' be a set of Υ'-structures. K' is $\mathrm{FO}_{\mathrm{static}}$*-reducible* to K iff there exists an $\mathrm{FO}_{\mathrm{static}}$ interpretation Φ such that for every Υ-structure I':

$$I' \in K' \text{ iff } \Phi I' \in K$$

Definition 4.52. A *circuit* over a vocabulary $\Upsilon \in$ voc is a structure C over the vocabulary $\Upsilon \in$ voc containing in addition to the function names in Υ only E, BOOL, STATIC, NEG, AND, OR, CONST1, CONST2, INP, OUT and $\tilde{f}$ for every $f \in \Upsilon$ (w.l.o.g., it is assumed that $\Upsilon \cap \{\tilde{f} : f \in \Upsilon\} = \emptyset$) such that:

- Active(C) contains the gates of the circuit.
- The functions consist of:
  - E is a 2-ary function describing the structure of the network ($E(x, y) =$ true iff the result of gate x is an input for gate y).
  - $\tilde{f}$ ($f \in \Upsilon_{\mathrm{stat}}$, so only static function names are considered here), BOOL, STATIC, NEG, AND, OR are unary functions which give the type of a gate in the circuit. $\tilde{f}$ indicates an application of f. BOOL stands for a gate introducing a truth value from Boole. STATIC indicates the introduction of an element from the background part by

use of a static function. NEG, AND, OR stand for the corresponding Boolean operations.

- INP is a unary (Boolean-valued) function, giving the input gates of the circuit. OUT is a unary (Boolean-valued) function representing the output gate. Here, circuits with only a single output gate are considered.
- CONST1 is a unary function, which assigns a truth value to those gates whose value of BOOL is true.
- CONST2 is a unary function, which assigns an element from the background to those gates whose value of STATIC is true.

Definition 4.53. Define $\mathrm{BCVP}_\Upsilon := (L^{\mathrm{BCVP}}_{1,\Upsilon}, L^{\mathrm{BCVP}}_{2,\Upsilon})$ where $L^{\mathrm{BCVP}}_{1,\Upsilon} := \{C : C$ is a circuit over Υ_{stat} and returns the value true for at least one input $\}$ and $L^{\mathrm{BCVP}}_{2,\Upsilon} := \{C : C$ is a circuit over $\Upsilon_{\mathrm{stat}}\}$.

Lemma 4.54. $\mathrm{BCVP}_\Upsilon \in \mathrm{NP}_{\mathrm{el}}$ *for all* $\Upsilon \in \mathrm{voc}$.

Proof. Obviously it is true that $\mathrm{BCVP} \in \mathrm{NP}_{\mathrm{el}}$. Just guess an input and then verify. The verification is possible in polynomial time. □

With the preceding definitions, one gets the following theorem:

Theorem 4.55. BCVP_Υ *is* $\mathrm{NP}_{\mathrm{el},\Upsilon}$*-hard via* $\mathrm{FO}_{\mathrm{static}}$*-reduction.*

Proof. Let $(L_1, L_2) \in \mathrm{NP}_\Upsilon$. According to theorem 4.49, there exists an existential second-order sentence (stat-Σ^1_1) $\varphi = \exists f_1 ... \exists f_n \psi$ such that
$L_2 = \{I : I \in L_2$ and $I \models \varphi\}$ and ψ is an $\mathrm{FO}_{\mathrm{static}}$-sentence over $\Upsilon \cup \{f_1, ..., f_n\}$, $\Upsilon \in \mathrm{voc}$.
Let $I \in L_2$. In the following, a circuit C, according to definition 4.52, will be constructed such that $I \in L_1$ iff $C \in L^{\mathrm{BCVP}}_{1,\Upsilon}$.
Because $\mathrm{size}(I)$ is finite, $\mathrm{Active}(I)$ is finite. So, w.l.o.g., assume that $\mathrm{Active}(I) = \{1, ..., n\}$ for some $n \in \mathbb{N}$.
W.l.o.g., one can assume that the formula φ does not contain any composed terms as $f(..., g(s), ...)$ where $f \in \{f_1, ..., f_n\} \cup \Upsilon$, $g \in \Upsilon_{\mathrm{dyn}}$. This can be assumed w.l.o.g. because, else, one can replace successively every atomic formula α, which contains such a term, by a formula $\exists t(g(x) = t) \wedge \alpha[g(x)/t]$ (where $\alpha[g(x)/t]$ is obtained from α by substituting all occurences of $g(x)$ by t.
Now, it will be described, how ψ can be interpreted as a circuit over the static part of Υ. The circuit consists of two parts:

1. all terms appearing in an atomic formula are computed
2. the circuit combines the truth values of the atomic formulae to yield the result

In the following, the functions of the circuit will be described.
The background structure of the circuit is the same as the one of I.
The active part of (the universe of) the circuit consists of all pairs $(\rho, \bar{a})$ where ρ is a subformula of ψ or a term occuring in ψ and $\bar{a} = a_1...a_k$ is an assignment to the variables of ρ.
Subformulae and terms in ψ will be called parts of ψ in the following.
So, define the following set:

$$B := \{(\rho, a_1...a_k) : \rho \text{ is a part of } \psi \text{ with } k \text{ free variables and } a_1, ..., a_k \in \text{Active}(I)\}$$

This is the active part of C.
$E((\rho, a_1...a_k), (\rho', b_1...b_l)) = 1$ iff all of the following conditions hold:

- ρ is a part of ρ'
- There is no part ρ'' of ρ' such that ρ is a part of ρ'
- For those variables which are free in both, ρ, ρ', the assignments $\bar{a}$ and $\bar{b}$ coincide.

The unary function BOOL assigns the value true to those $(\rho, \bar{a})$ such that ρ is an atom of the type $R(t_1, ..., t_j)$ where R is a function of type $f : (\text{Active}(I))^r \rightarrow \text{Active}(I)$ tagged relational or ρ is an equality.
The unary function STATIC is true for those $(\rho, \bar{a})$ such that ρ is a static constant or a term $f(t_1, ..., t_j)$ where $f \in \Upsilon_{\text{stat}}$.
The function INP maps those gates $(\rho, \bar{a})$ to true where ρ is a part of ϕ of type $f(t_1, ..., t_j)$ where f is quantified in φ (the inputs of the circuit are represented by this).
The function OUT assigns true to ψ (for ψ there is no variable assignment because ψ does not contain any free variable).
A function $\tilde{f}$ assigns true to a gate $(\rho, \bar{a})$ iff ρ is a term of the form $f(t_1...t_j)$.
The unary function OR assigns true to a gate $(\rho, \bar{a})$ if either ρ is the disjunction of two subformulae or $\rho = \exists t \rho'$ for a subformula ρ'.
The unary function AND assigns true to a gate $(\rho, \bar{a})$ if either ρ is the conjunction of two subformulae or $\rho = \forall t \rho'$ for a subformula ρ'.
The unary function NEG assigns true to a gate $(\rho, \bar{a})$ if ρ is the negation of a subformulae.
CONST1 is given by a $\text{FO}_{\text{static}}$-formula over Υ assigning to each $(\rho, \bar{a})$ with $\text{BOOL}((\rho, \bar{a})) =$ true the truth value of the corresponding relation. CONST2 does the same for $(\rho, \bar{a})$ with $\text{STATIC}((\rho, \bar{a})) =$ true.
The described circuit has the intended properties. I.e., $C \in \text{BCVP}_\Upsilon$ iff $I \in L$.
The size of B is polynomially bound by size(I) because the number of subformulae in ψ is constant (for a given set of of structures) and there are at most polynomially many different assignments to the free variables of any such subformula. □

This is a result rather similar to the one of [7] about algebraic circuits but in [7], the circuits are built over $\mathbb{R}$ and here it is the background part of the considered structure.
Because of lemma 4.47, this reduction is also a polynomial time reduction.
One important difference between rule-non-determinism resp. determinism and the element-non-determinism is that programs with only rule-non-determinism resp. determinism only deal with accessible elements of the structure. If element-non-determinism is used, this is not true. The guess-element rule takes any element of the living universe (satisfying a given condition), it might be accessible or inaccessible. So, for a program with only rule-non-determinism resp. determinism, the background part of the input structure is only important as far as it is accessible. One can even go farer: Only a finite part of the strucure is of importance for those programs, because the program can be represented by a finite string and only finitely many elements can be accessed via this program within a finite. So, it is not too surprising that in the following two theorems the background structure (and its potential infiniteness) is not that important anymore.

Definition 4.56. A *Boolean expression* is a labeled tree and the propositional variables are ordered by a successor function Succ and a nullary function firstvar giving the first variable. Furthermore, the unary function name $\mathcal{I}$ is not contained in the vocabulary of a Boolean expression and it is interpreted as undef for all elements of the universe.

Definition 4.57. Define $\mathrm{SAT} := (L_1^{\mathrm{SAT}}, L_2^{\mathrm{SAT}})$ where
$L_1^{\mathrm{SAT}} := \{C : C \text{ is a satisfiable Boolean expression}\}$ and
$L_2^{\mathrm{SAT}} := \{C : C \text{ is a Boolean expression}\}$

Theorem 4.58. SAT *is* $\mathrm{NP}_{\mathrm{rule}}$*-complete via polynomial macro-time-reduction.*

Proof. First, it will be shown that $\mathrm{SAT} \in \mathrm{NP}_{\mathrm{rule}}$. To show this, an algorithm will be described which decides SAT within rule-non-deterministic, polynomial macro time.
With the following, an interpretation $\mathcal{I}$ will be guessed:

```
guess-rule
   I(firstvar) := false
   I(firstvar) := true
endguess-rule
firstvar := Succ(firstvar)
```

Afterwards, $\mathcal{I}$ maps every variable (or to be more precise every node in the tree labeled with a variable) in the Boolean expression to a Boolean value (the nodes in the tree labled with $\wedge$, $\vee$ or $\neg$ are mapped to undef). Now, it will be verified, that $\mathcal{I}$ gives a model for the Boolean expression. This is done in the

obvious way.
Second, it will be proven that SAT is $\mathrm{NP}_{\mathrm{rule}}$-hard via polynomial time-reduction. Time will be measured by macrosteps. So, for a rule-non-deterministic ASM $\mathcal{A} = (Q, \mathrm{Input})$ over $\Upsilon \in$ voc, such that there exists a polynomial p with $\mathrm{NTIME}^{\mathrm{macro}}_{\mathrm{rule}}(Q, J) \leq p(\mathrm{size}(K))$ for all $K \in$ Input, and a structure $I \in$ Input, a Boolean expression $\varphi_{\mathcal{A},I}$ will be constructed such that $\varphi_{\mathcal{A},I} \in L_1^{\mathrm{SAT}}$ iff $I \in \mathrm{Acc}(\mathcal{A})$.
In the following, the Boolean expression $\varphi_{\mathcal{A},I}$ will be constructed.
W.l.o.g., it can be assumed that Q is of the form (according to lemma 2.7 and the remark in section 4.7):

```
import v_1
  .
  .
  .
import v_k
  guess-element w_1 :  Boole(w_1)
    .
    .
    .
  guess-element w_l :  Boole(w_l)
    block
      if c_1 then R_1 endif
        .
        .
        .
      if c_m then R_m endif
    endblock
  endguess-element
    .
    .
    .
  endguess-element
endimport
  .
  .
  .
endimport
```

where R_i, $i \in \{1, ..., m\}$ are updates.
Furthermore, w.l.o.g., one can assume that the conditions in the if-clauses are conjunctions of equalities because `if` $\alpha \vee \beta$ `then` R `endif` is equivalent to `if` α `then` R `endif; if` β `then` R `endif` and $\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$.
If the number of macrosteps of the original program is bound by a polynomial,

then the number of macrosteps of the resulting program is also bound by a polynomial.
Let accessible(t) be the subset of the universe of the Υ-structure I which contains all elements that are accessible until time t (inclusively). So, only these elements are relevant for the computation until that time.
This set can be as follows:

$$\text{accessible}(t) = \text{accessible}(t, \max\{\text{depth}(s) : s \text{ is a term appearing in } Q\})$$

where accessible(t, l) is defined inductively:

- $\text{accessible}(0,0) = \{c^I : c \in \Upsilon \text{ is a nullary function name }\}$
- $\text{accessible}(t+1,l) = \{f^I(x_1,...,x_r) : x_1,...,x_r \in \text{accessible}(t,l) \wedge f \in \Upsilon\}$
- $\text{accessible}(t,l+1) = \{f^I(x_1,...,x_r) : x_1,...,x_r \in \text{accessible}(t,l) \wedge f \in \Upsilon\}$

Let disposible(t) := accessible(t) $\cup \{v_{i,t'} : 0 \leq t' \leq t, i \in \{1,...,k\}\}$. This is the set of all elements accessible within t macrosteps and the variables representing the imported elements until macro time t.
$\varphi_{\mathcal{A},I}$ has the following appearance: $\varphi_{\mathcal{A},I} = \text{START} \wedge \text{COMPUTE} \wedge \text{ACCEPT}$ and uses one kind of variables $X_{s,b,t}$:

$X_{f,a_1,...,a_r,b,t}$: $f \in \Upsilon_{\text{dyn}}$ is an r-ary function name and $a_1,...,a_r,b \in \text{disposible}(p(\text{size}(I)))$. $X_{f,a_1,...,a_r,b,t}$ is true iff the content of the location $(f,a_1,...,a_r)$ is b at time t.

START : START describes the input structure I. I.e., the elements accessed while executing the program and the interpretations of the function names restricted to this part of the structure.
More precisely, START has the following form: START is the conjunction of all $X_{f,a_1,...,a_r,b,0}$ such that $f^I(a_1,...,a_r) = b$, $f \in \Upsilon$, $r = \text{arity}(f)$, $a_1,...,a_r,b \in \text{disposible}(p(\text{size}(I)))$. If $a_j \in \{v_{i,t'} : 0 \leq t' \leq p(\text{size}(I)), i \in \{1,...,k\}\}$ for an $j \in \{1,...,r\}$ then $f^I(a_1,...,a_r) = \text{undef}$ (because $v_{i,t}$ represents the i-the element imported at time t.

COMPUTE : COMPUTE describes an accepting of Q on I.
First define Eval(s,b,t) for a term s constructed by use of function names from Υ and elements from $\{v_{i,t'} : 0 \leq t' \leq t, i \in \{1,...,k\}\}$, $b \in \text{disposible}(t)$, t gives the time. Eval(s,b,t) is defined by induction on s:

- if $s = c$ for a nullary function name $c \in \Upsilon$ or $s = v$ for $v \in \{v_{i,t'} : 0 \leq t' \leq t, i \in \{1,...,k\}\}$ then $\text{Eval}(s,b,t) = X_{s,b,t}$.
- if $s = f(s_1,...,s_r)$ for $f \in \Upsilon$, $\text{arity}(f) = r$ and s_1, ..., s_r are terms constructed by use of function names from Υ and elements from $\{v_{i,t'} : 0 \leq t' \leq t, i \in \{1,...,k\}\}$ then

$\mathrm{Eval}(s, b, t) = \bigvee_{b_1,...,b_r \in \mathrm{disposible}(t)} \mathrm{Eval}(s_1, b_1, t) \wedge ... \wedge \mathrm{Eval}(s_r, b_r, t) \wedge X_{f,b_1,...,b_r,t}$

Consider the block of if-clauses inside the program Q. For every $M \in \mathcal{P}(\{1, ..., n\})$ define a Boolean expression $\psi_{M,t}$: For every if-clause `if` c `then` $f(u_1, ..., u_{\mathrm{arity}(f)}) := g(v_1, ..., v_{\mathrm{arity}(g)})$ introduce the following:

$$(\bigwedge_{i \in M} \mathrm{translation}(c_i, t) \wedge \bigwedge_{i \notin M} \neg \mathrm{translation}(c_i, t)) \rightarrow \varphi_{M,t}$$

$\varphi_{M,t}$ describes the updates caused by those if-clauses i such that $i \in M$. Other changes of the contents of the locations are avoided. The updates can be grouped by the function names given in the updated locations. Let $M_f := \{f(s_1^1, ..., s_r^1) := s_0^1, ..., f(s_1^r, ..., s_r^r) := s_0^r\}$ be the set of all updates which update a location location with function name f and belong to the i-th if-clause for an $i \in M$. So, $\varphi_{M,t} = \bigwedge_{f \in \Upsilon_{\mathrm{dyn}}} \varphi_{M,f,t}$. In $\varphi_{M,f,t}$ only those updates are considered which deal with f. Now, $\varphi_{M,f,t}$ will be given:

$$\begin{aligned}
\varphi_{M,f,t} = & \\
& \bigwedge_{f(s_1,...,s_r):=s_0 \in M_f} \bigwedge_{b \in \mathrm{disposible}(t)} (\mathrm{Eval}(s_0, b, t) \rightarrow \mathrm{Eval}(f(s_1, ..., s_r), b, t+1) \\
\wedge & \bigwedge_{b_1,...,b_r \in \mathrm{disposible}(p(\mathrm{size}(I)))} (\bigwedge_{f(t_1,...,t_r):=s \in M_f} (\neg \bigwedge_{i \in \{1,...,r\}} \mathrm{Eval}(s_i, b_i, t)) \\
& \rightarrow (\bigwedge_{b \in \mathrm{disposible}(p(\mathrm{size}(I)))} X_{f,b_1,...,b_r,b,t} \rightarrow X_{f,b_1,...,b_r,b,t+1}))
\end{aligned}$$

$\mathrm{translation}(c, t)$ is the translation of the guard c at time t. It is defined inductively as follows:

- if c is an equality $s_1 = s_2$ where s_1 and s_2 are terms then $\mathrm{translation}(c, t) = \mathrm{Eval}(s_1, t) \leftrightarrow \mathrm{Eval}(s_2, t)$.
- if c is a negation $\neg c'$ then $\mathrm{translation}(c, t) = \neg \mathrm{translation}(c', t)$
- if c is a conjunction $c_1 \wedge c_2$ then $\mathrm{translation}(c, t) = \mathrm{translation}(c_1, t) \wedge \mathrm{translation}(c_2, t)$
- if c is a disjunction $c_1 \vee c_2$ then $\mathrm{translation}(c, t) = \mathrm{translation}(c_1, t) \vee \mathrm{translation}(c_2, t)$

Now, the i-the guess-rule-rule is considered. It has the form

```
guess-rule
   R1
   R2
endguess-rule
```

where R and S have already been translated at time t (the results are translation(R,t) and translation(S,t)). Then the the translation of the guess-rule rule at time t is translation$(R,t) \vee$ translation(S,t).
For Q and time t let translation(Q,t) be the Boolean eXpression gained as described above. Then

$$\text{COMPUTE} = \bigwedge_{t \in \{0,\ldots,p(\text{size}(I))} \text{translation}(Q,t) \wedge \text{Exclude}$$

where Exclude is the following Boolean expression:

$$\bigwedge_{t \in \{0,\ldots,p(\text{size}(I))} \bigwedge_{f \in \Upsilon_{\text{dyn}}} \bigwedge_{a_1,\ldots,a_{\text{arity}(f)}, b \in \text{disposible}(t)}$$
$$X_{f,a_1,\ldots,a_{\text{arity}(f)},b,t} \rightarrow \bigwedge_{c \in \text{disposible}(t), c \neq b} \neg X_{f,a_1,\ldots,a_{\text{arity}(f)},c,t}$$

Exclude has to be introduced because a location can only have one value. Exclude ensures this property.

ACCEPT : ACCEPT ensures that the last state is a final and accepting one. This expression is the most simple one of the three parts:
$\bigvee_{t' \leq p(\text{size}(I))} (X_{\text{Output},\text{true},t'} \wedge X_{\text{Mode},\text{false},t'})$

As one can easily see, the size of $\varphi_{\mathcal{A},I}$ is only polynomial in the size of the input (because the number of subterms of a term is only linear in the length of the term) and so is the time to construct it. □

Theorem 4.59. SAT *is* $\text{NP}_{\text{el,finite}}$*-complete via polynomial macro-time-reduction.*

Proof. First, it will be proven that SAT $\in \text{NP}_{\text{el,finite}}$. To prove this, an algorithm will be described which decides SAT in element-non-deterministic, polynomial macro time. For this, it is assumed that every input structure contains additionally an 1-ary function name $\mathcal{I}$ in its vocabulary which is initially undefined for all nodes (of the tree which represents the Boolean expression). With the following, an interpretation $\mathcal{I}$ will be guessed:

```
block
  guess-element v :  Boole(v)
    I(firstvar) := v
  endguess-rule
  firstvar := Succ(firstvar)
endblock
```

Afterwards, $\mathcal{I}$ maps every variable (or more precisely every node in the tree labeled with an variable)in the Boolean expression to a Boolean value (the nodes

in the tree labled with $\wedge$, $\vee$ or $\neg$ are mapped to undef). Now, it will be verified, that $\mathcal{I}$ gives a model for the Boolean expression. This is done in the obvious way.
The $\mathrm{NP}_{\mathrm{el,finite}}$-hardness of SAT can be shown elementarily by constructing a Boolean expression rather similar to the formula constructed in the proof of theorem 4.49.
By use of the theorems already shown and their proofs, the proof of this theorem can be done much shorter and easier. In this proof, theorem 4.49, theorem 4.55 and their proofs will be used.
The construction of the circuit in the proof of theorem 4.55 continues with the sentence constructed in the proof of theorem 4.49.
In this special case, the circuit has only the functions E, BOOL, NEG, AND, OR, CONST1, $\vee$, $\wedge$, $\neg$. BOOL maps every gate to true, CONST1 gives the truth values at every gate. The functions STATIC and CONST2 are unnecessary because there are notstatic functions besides the logical ones. So, this circuit C can directly be translated to a Boolean expression α_C and the following holds: α_C is satisfiable iff C returns the value true for at least one input.
The size of the Boolean expression is polynomially bound by the size of the input structure (following the construction).

Another possibilty to prove the hardness is construct a Boolean expression as in the proof of theorem 4.58 with the following difference:
Instead of the disjunction of the translations of the two rules in the guess-rule rule, construct the disjunction of all active elements at that time. The guard of a guess-element-rule can be translated as the guard of an if-clause. $\square$

Consider the sets of structures (coded as strings) which are NP-complete in the usual sense, e.g. Vertex Cover, Clique, Hamilton Cycle, ... Using the same reductions as for Turing machines, one gets that the pairs consisting of the respective complete set and the set of all those structures, which might have the property, are also $\mathrm{NP}_{\mathrm{rule}}$-complete and $\mathrm{NP}_{\mathrm{el,finite}}$-complete via polynomial time reduction. E.g., $(\{G : G \text{ is a graph and contains a Hamilton cycle}\}, \{G : G \text{ is a graph}\})$ is $\mathrm{NP}_{\mathrm{rule}}$-complete via polynomial time reduction.

Definition 4.60. A *Boolean Horn-formula* is a Boolean expression $\psi = \bigwedge_{i\in I} \bigvee_{j\in J} Y_{i,j}$ such that I and J are finite sets of indexes and every disjunction $\bigvee_{j\in J} Y_{i,j}$ contains at most one positive literal.
This leads to the following: Define $\mathrm{HORN-SAT} := (L_1^{\mathrm{HORN-SAT}}, L_2^{\mathrm{HORN-SAT}})$ where
$L_1^{\mathrm{HORN-SAT}} := \{\psi : \psi \text{ is a satisfiable Boolean Horn-formula}\}$ and $L_2^{\mathrm{HORN-SAT}} := \{\psi : \psi \text{ is a Boolean Horn-formula}\}$

Note that Boolean Horn-formulae can also be written as a conjunction of implications by use of the following two equivalences:

$$\begin{aligned} \neg X_1 \vee ... \vee \neg X_k \vee X &\equiv (X_1 \wedge ... \wedge X_k) \to X \\ \neg X_1 \vee ... \vee \neg X_k &\equiv (X_1 \wedge ... \wedge X_k) \to \text{false} \end{aligned}$$

Definition 4.61. $\Upsilon_{\text{in}}, \Upsilon_{\text{out}}, \Upsilon_{\text{work}} \in \text{voc}$ such that the intersection of two such vocabulary contains only the obligatory function names and function names tagged as static. Furthermore, $(\Upsilon_{\text{out}} \cup \Upsilon_{\text{work}} \in \text{voc}) - \Upsilon_{\text{in}} = \{f_1, ..., f_n\}$
Let (L_1, L_2) be a pair of sets of Υ_{in}-structures and (L_1', L_2') be a pair of sets of Υ_{oput}-structures. Then (L_1, L_2) is logspace-reducible to (L_1', L_2') if there exists an ASM $\mathcal{A} = (P, \text{Exp}(L_2, f_1, ..., f_n))$ and a constant $c_{\mathcal{A}}$ such that A logspace-reduction is given by a reduction satisfying the following conditions for every input structure I:

- P halts on every $I \in \text{Exp}(L_2, f_1, ..., f_n)$.
- The elements of $\Upsilon_{\text{in}} - \{\text{Mode}, \text{Output}\}$ are read-only.
- The elements of $(\Upsilon_{\text{out}} - \{\text{Mode}, \text{Output}\})_{\text{dyn}}$ are write-only.
- Let $I \in \text{Exp}(L_2, f_1, ..., f_n)$. Then in no state in the run of P, the structure restricted to the elements Υ_{work} has a size greater than $c_{\mathcal{A}} \cdot \log(\text{size}(I))$ on I, the structure given by the elements structure given by the
- For the last structure J in the run of P on $I \in \text{Exp}(L_2, f_1, ..., f_n)$ the following is valid: J restricted to the elements of $(\Upsilon_{\text{out}}$ is in L_1' iff I restricted to the elements of $(\Upsilon_{\text{i}}$ is in L_1.

Theorem 4.62. HORN $-$ SAT *is* P*-complete via logspace-reduction.*

Proof. A Boolean Horn-formula is unsatisfiable iff $\square$ (the empty clause) can be derived from the corresponding set of clauses by unit-resolution. This can be checked in deterministic polynomial time. Therefore, HORN $-$ SAT $\in$ P.
Now, prove that HORN-SAT is P-hard via logspace-reduction. A deterministic ASM-program is a special case of an ASM-program with only rule-non-determinism.
Let $(L_1, L_2) \in \text{P}$. So, there exists a (deterministic) ASM $\mathcal{A} = (Q, L_2)$ and a polynomial p such that $\text{TIME}^{\text{macro}}(Q, I) \leq p(\text{size}(I))$ for all $I \in L_2$ and $\text{Acc}(Q) = L_1$. Take the reduction from the proof of theorem 4.58 with two difference:

1. Instead of the normal form from lemma 2.7, one gets the normal form from lemma 2.5.
2. The program does not use an guess-rule and therefore, the translation for such a rule has to be left away.

Therefore, the result of this reduction can easily be transformed into a Boolean Horn-formula.
The discribed reduction is a logspace-reduction. □

As a consequence, all pairs of sets of structures resulting from those sets of structures (resp. their encodings), which are P-complete via logspace-reduction for Turing machines, as for $\mathrm{NP}_{\mathrm{rule}}$, are also P-complete via logspace-reduction for ASMs. This can be proven by using the same reductions as for Turing machines. E.g., the circuit value problem one such set.

Chapter 5

Space Complexity

5.1 Deterministic Space

The size of an Υ-structure I has already been defined in definition 3.1. This is the basis of the following chapter about space complexity.

Definition 5.1. Given an ASM-program P over $\Upsilon \in \text{voc}$ and an Υ-structure $I \in \text{Inputstates}(P)$. Let $s_0 s_1 ... s_k$, $I = s_0$, be the run of P on I as a sequence of states and s_k is a final state. Then the (deterministic) space is given by $\text{SPACE}(P, I) = \max\{\text{size}(s_j) : j = 0, ..., k\}$.

Definition 5.2. The (deterministic) space time needed by an ASM-program P over Υ for structures from $L \subseteq \text{Inputstates}(P)$ of size n is given by

$$\text{SPACE}(P, L, n) := \sup\{\text{SPACE}(P, I) : I \in L \text{ such that } \text{size}(I) = n\}$$

Definition 5.3. Fix $\Upsilon \in \text{voc}$.
$\text{SPACE}_\Upsilon(f(n)) := \{(L_1, L_2) : L_1, L_2$ are sets of Υ-structures of finite size, $L_1 \subseteq L_2$, and there exists a (deterministic) ASM $\mathcal{A} = (P, L_2)$ over Υ such that $L_1 = \text{Acc}(\mathcal{A})$ and $\text{SPACE}(P, \text{size}(I)) \leq f(\text{size}(I))$ for all $I \in L_2\}$

Definition 5.4. $\text{SPACE}(f(n)) := \bigcup_{\Upsilon \in \text{voc}} \text{SPACE}_\Upsilon(f(n))$

The assumptions of the following definition are the same as in the definitions of *macro time constructible* and *micro time constructable.*

Definition 5.5. Let $\Upsilon = \{\text{Mode}, \text{Output}, \text{true}, \text{false}, \text{undef}, \text{Boole}, \neg, \vee, \wedge, =\} \cup \{\text{firstActive}, \text{lastActive}, \text{SuccActive}\} \in \text{voc}$ such that firstActive and lastActive are dynamic nullary function names and SuccActive is a dynamic unary function name. The rest of the vocabulary (the obligatory part) is as already described. Let $\text{Order} := \{I : I$ is an Υ-structure of finite size, SuccActive^I is a successor function in $\text{Active}(I)$ which orders $\text{Active}(I)$ totally linearly, firstActive^I gives the first element and lastActive^I gives the last element, $\text{Mode}^I = \text{true}, \text{Output}^I = \text{false}\}$.

A function $f : \mathbb{N} \to \mathbb{N}$ is called *macro time constructible* if there is an (deterministic) ASM $\mathcal{A} = (P, \mathrm{Order})$ such that for every $I \in \mathrm{Order}$, $\mathrm{SPACE}(P, I) = f(|\mathrm{Active}(I)|)$ and the last structure of every run satisfies the conditions for being an element of Order apart from the contents of Output and Mode.

Lemma 5.6. *Let* $\Upsilon \in \mathrm{voc}$ *be an order-solvent and copy-solvent vocabulary.* f *be a space constructible function and* $f(n) \geq n$. *Let* $\mathcal{A} = (P, \mathrm{Input})$ *be a (deterministic) ASM such that* $\mathrm{Input} \subseteq \mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyCopy}_\Upsilon$ *and* $\mathrm{SPACE}(P, I) \leq f(\mathrm{size}(I))$ *for all accepted structures* $I \in \mathrm{Input}$.
Then there exists a (deterministic) ASM $\mathcal{A}' = (P', \mathrm{Input})$ *such that* $\mathrm{Acc}(\mathcal{A}') = \mathrm{Acc}(\mathcal{A})$ *and* $\mathrm{SPACE}(P', I) \leq 3 \cdot f(\mathrm{size}(I))$ *for all* $I \in \mathrm{Input}$.

Proof. Let $\Upsilon \in \mathrm{voc}$ be an order-solvent and copy-solvent vocabulary. Given a (deterministic) ASM $\mathcal{A} = (P, \mathrm{Input})$ over Υ. Because f is space constructible, an ASM $\mathcal{B} = (Q, \mathrm{ActiveOrder}_\Upsilon)$ exists satisfying the property of definition 5.5.
Construct a new program P' that works as follows: First, construct a disjoint structure from Order which contains $\mathrm{size}(I)$ as already described earlier. Then let run $\mathcal{B}$ this structure. This leads to a structure $J \in \mathrm{Order}$ of size $f(size(I))$.
The unary function copy is used to build a partial mapping from Order to I. To every active element of I as many elements of J are mapped as there are locations which contain the active element. So, the basic idea is a mapping from the elements to locations whose content is inequal to undef.
This mapping can be preserved for every state of the run by resetting copy according to the executed updates.
Such a mapping exists iff the structure disjoint from J has a size $\leq f(\mathrm{size}(I))$.
If P stops and uses more than $f(\mathrm{size}(I))$ space (i.e., in the rejecting run of P on I a structure appears of size $\geq f(\mathrm{size}(I))$), then P' stops and rejects.
If P never stops and produces no structure of size $\geq f(\mathrm{size}(I))$, then Q never stops.
Obviously (i.e., this is a direct consequence of the construction) $\mathrm{Acc}(Q) = \mathrm{Acc}(P)$ and $\mathrm{SPACE}(Q, I) \leq 3 \cdot f(\mathrm{size}(I))$ for every Υ-structure I. □

No bound can be calculated how many states can be reached within a given limit of determinstic space. E.g. remember the set of structures L_{inf} defined at the beginning of chapter 4. Consider the following program checking whether a given structure (with vocabulary and background as already described) is in L_{inf} by trying all natural numbers starting from 0 until one is found satisfying the condition of the definition L_{inf}.

```
block
    if Mode = true then Mode := 0 endif
    if Mode ≠ true then
     block
       if f(Mode) ≠ undef then
         block
           Mode := false
           Output := true
         endblock
       endif
       if f(Mode) = undef then Mode := Succ(Mode) endif
     endblock
  endif
```

The following lemma holds only in the special case. This has already been proven by use of L_{inf}.

Lemma 5.7. *Let* $\Upsilon \in \mathrm{voc}_{\mathrm{finite}}$ *be an order-solvent and copy-solvent vocabulary and* f *be a space constructible function and* $f(n) \geq n$.
Let $\mathcal{A} = (P, \mathrm{Input})$ *be a (deterministic) ASM such that* $\mathrm{Input} \subseteq \mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyCopy}_\Upsilon$ *and* $\mathrm{SPACE}_{\mathrm{finite}}(P, I) \leq f(\mathrm{size}(I))$ *for all accepted structures* $I \in \mathrm{Input}$. *Then there exists a (deterministic) ASM* $\mathcal{A}' = (P', \mathrm{Input})$ *such that* $\mathrm{Acc}(\mathcal{A}') = \mathrm{Acc}(\mathcal{A})$, $\mathrm{SPACE}_{\mathrm{finite}}(P', I) \leq 9 \cdot f(\mathrm{size}(I))$ *for all* $I \in \mathrm{Input}$ *and* P' *halts on every* $I \in \mathrm{Input}$.

Proof. Because of lemma 5.6, w.l.o.g., an ASM (P, L_2) can be considered such that $\mathrm{SPACE}(P, J) \leq 3 \cdot f(\mathrm{size}(J))$ for $J \in L_2$.
Now, construct an ASM (P', L_2) halting on all elements of L_2 and satisfying $\mathrm{SPACE}(P', J) \leq 9 \cdot f(\mathrm{size}(J))$ for $J \in L_2$.
Consider an arbitrary structure $I \in L_2$. No structure appearing in a run of P starting with input I has a size greater than $f(\mathrm{size}(I))$. So, first, the number of structures of size equal to $f(size(I))$ over a given universe is computed.
Given a structure $I \in L_2$. Define $z := 3 \cdot f(\mathrm{size}(I))$. For every structure J with $\mathrm{size}(J) \leq 3 \cdot f(\mathrm{size}(I))$

- J contains at most z elements.
- For every function name $g \in \Upsilon$ there are at most z elements (as elements of $(\mathrm{Living}(J))^{\mathrm{arity}(g)}$) and the range contains at most z elements.

Define $y := |\Upsilon|$. So, for every $g \in \Upsilon$ there are at most $z^z = 2^{z \cdot \log(z)}$ possibilities to interprete it in structure a structure J with $\mathrm{size}(J) = z$. So, there are at most $(2^{z \cdot \log(z)})^y = 2^{y \cdot z \cdot \log(z)}$ structures J with $\mathrm{size}(J) = z$.
Therefore, there are $2^{c_P \cdot f_1(\mathrm{size}(I)) \cdot \log(f_1(\mathrm{size}(I)))}$ (where c_P depends only on P) of size $\leq f_1(\mathrm{size}(I))$. So, P stops after $2^{c_P \cdot f(\mathrm{size}(I)) \cdot \log(f_1(\mathrm{size}(I)))}$ macrosteps or never.

Now, introduce a mechanism such that P' stops itself after $2^{c_P \cdot f(\mathrm{size}(I)) \cdot \log(f(\mathrm{size}(I)))}$ macrosteps.
As already done in some of the preceding proofs, build a disjoint structure as described in definition 5.5 (i.e., an element of Order). Because $3 \cdot f$ is space constructible (this is a direct consequence of space constructibility of f), there exists an ASM $\mathcal{B} = (Q, \mathrm{Order})$ satisfying the condition from definition 5.5. This leads to a disjoint, ordered structure of size equal to $3 \cdot f(\mathrm{size}(I))$. Now, simulate the work of P on I and in parallel to this, count the number of macrosteps as follows:
Mark the first element of the order, follow the chain given by the order 2^{c_P} times starting from the first element. Move the mark to next element and again, follow the chain 2^{c_P} times starting from the first element. Repeat this until the mark has reached the last element of the order. Then follow the chain 2^{c_P} times starting from the first element and stop.
This corresponds to a counter with basis 2^{c_P}. □

5.2 Non-Deterministic Space

Definition 5.8. Given a non-deterministic ASM-program P (either element-non-deterministic or rule-non-deterministic) and an Υ-structure $I \in \mathrm{Inputstates}(P)$. Let $M = \{s_0 s_1 ... s_k : I = s_0$ and $s_0 s_1 ... s_k$ is a run of P on I as a sequence of states and s_k is a final state$\}$. Then the non-deterministic space is $\mathrm{NSPACE}_i(P, I) = \sup\{\mathrm{size}(s_j) : j \in \{0, ..., k\}, s_0 s_1 ... s_k \in M\}$.

Definition 5.9. For $i \in \{\mathrm{el}, \mathrm{rule}\}$ the non-deterministic space needed by an ASM-program P over Υ for structures from $L \subseteq \mathrm{Inputstates}(P)$ of size n is given by

$$\mathrm{SPACE}_i(P, L, n) := \sup\{\mathrm{SPACE}_i^{\mathrm{macro}}(P, I) : I \in L \text{ such that } \mathrm{size}(I) = n\}$$

Definition 5.10. Fix $\Upsilon \in \mathrm{voc}$. Let $i \in \{\mathrm{el}, \mathrm{rule}\}$.
$\mathrm{SPACE}_{i,\Upsilon}(f(n)) := \{(L_1, L_2) : L_1, L_2$ are sets of Υ-structures of finite size, $L_1 \subseteq L_2$, and there exists a non-deterministic ASM $\mathcal{A} = (P, L_2)$ of kind i over Υ such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{SPACE}(P, \mathrm{size}(I)) \leq f(\mathrm{size}(I))$ for all $I \in L_2\}$

Definition 5.11. Let $i \in \{\mathrm{el}, \mathrm{rule}\}$.
$\mathrm{NSPACE}_i(f(n)) := \bigcup_{\Upsilon \in \mathrm{voc}} \mathrm{NSPACE}_{i,\Upsilon}(f(n))$

5.3 Determinism And Non-Determinism

Definition 5.12. Let pol be the set of all polynomials, $\Upsilon \in$ voc. Define the following complexity classes:

$$\begin{aligned}
\mathrm{PSPACE}_\Upsilon &:= \bigcup_{p \in \mathrm{pol}} \mathrm{SPACE}_\Upsilon(p(n)) \\
\mathrm{PSPACE} &:= \bigcup_{p \in \mathrm{pol}} \mathrm{SPACE}(p(n)) \\
\mathrm{NPSPACE}_{i,\Upsilon} &:= \bigcup_{p \in \mathrm{pol}} \mathrm{NSPACE}_{i,\Upsilon}(p(n)) \\
\mathrm{NPSPACE}_{i} &:= \bigcup_{p \in \mathrm{pol}} \mathrm{NSPACE}_{i}(p(n))
\end{aligned}$$

5.4 Relations Between Complexity Classes

The following theorem is a consequence of the Diagonalization Theorem.

Lemma 5.13. *Let $\Upsilon \in \mathrm{voc}_{\mathrm{finite}}$ be a diagonalization-solvent, order-solvent and graph-solvent vocabulary. Let $f_2 : \mathbb{N} \to \mathbb{N}$ be a space constructible and $f_1 : \mathbb{N} \to \mathbb{N}$. If $f_2(n) \geq n$ and $f_1 = o(f_2)$ then*

$$\begin{aligned}
&(\mathrm{SPACE}_\Upsilon(3 \cdot f_2(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon)) \\
- \ &(\mathrm{SPACE}_\Upsilon(f_1(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon)) \neq \emptyset
\end{aligned}$$

Proof. It will be proven that for every diagonalization-solvent vocabulary, order-solvent and copy-solvent vocabulary $\Upsilon \in$ voc, $\mathrm{SPACE}_\Upsilon(f_2(n))$ allows diagonalization over $\mathrm{SPACE}_\Upsilon(f_1(n))$ concerning $\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon$. From the diagonalizition theorem, one can then conclude that this lemma holds.
Consider $(L_1, L_2) \in \mathrm{SPACE}_\Upsilon(f_1(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon))$. So, there exists an ASM $\mathcal{A} = (P, L_2)$ such that $L_1 = Acc(\mathcal{A})$ and for $I \in L_2$, $\mathrm{SPACE}(P, I) \leq f_1(\mathrm{size}(I))$. Because f_2 is space constructible, there is an ASM $\mathcal{B} = (Q, \mathrm{Order})$ satisfying the conditions from definition 5.5. given by a program Q over Υ such that for every Υ-structure I, $\mathrm{SPACE}(P, I) = f_2(size(I))$.
In the following, an ASM $\mathcal{A}_{\mathrm{Diag}} = (\mathrm{Diag}, (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon))$ will be constructed which satisfies the conditions of definition 3.14. Diag works as follows:
Diag interprets every input I as an ASM $\mathcal{A}$, i.e., $P = I(\Upsilon, \mathcal{A})$ (it is assumed that I encodes a program else it is rejected). Diag simulates the work of P on I. As in the proof of lemma 5.6 it is controled whether more space than size(I) is used. If P needs more than $f(\mathrm{size}(I))$ space then Diag rejects. If P accepts within the

given limits, then Diag rejects. If P rejects within the given limits, then Diag accepts.
Because of lemma 5.7, one can suppose that Diag halts on every structure from $\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon$.
$\mathcal{A}_{\mathrm{Diag}}$ satisfies the conditions of definition 3.14, because: If $\mathrm{SPACE}(P, I) \leq f_1(\mathrm{size}(I))$, then P is simulated completely on $I(\Upsilon, \mathcal{A})$. And for those P it is true that $I(\Upsilon, \mathcal{A}) \in \mathrm{Acc}(\mathcal{A}_{\mathrm{Diag}})$ iff $I(\Upsilon, \mathcal{A}) \notin \mathrm{Acc}(P)$ □

The factor rests because one has no compression theorem for space.
The following corollary is a direct consequence of the previous theorem.

Corollary 5.14. *Let $\Upsilon \in \mathrm{voc}_{\mathrm{finite}}$ be a diagonalization-solvent, order-solvent and copy-solvent vocabulary. Let $f_2 : \mathbb{N} \to \mathbb{N}$ be a space constructible and $f_1 : \mathbb{N} \to \mathbb{N}$. If $f_2(n) \geq n$ and $f_1 = o(f_2)$ then $\mathrm{SPACE}_\Upsilon(4 \cdot f_2(n)) - \mathrm{SPACE}_\Upsilon(f_1(n)) \neq \emptyset$.*

The following theorem holds only in the special case but not in the general case, because the guess-element rule allows to guess an element from the whole living part (and so also from the background), which might be infinite in the general case.
The proposition of the follwing theorem is similar to the one of the theorem of Savitch in classical complexity theory.

Theorem 5.15. *Let $\Upsilon \in \mathrm{voc}_{\mathrm{finite}}$ be an order-solvent and graph-solvent vocabulary and f be a space constructible function. Let $i \in \{\mathrm{el}, \mathrm{rule}\}$.
Then*

$$\begin{aligned} & \mathrm{NSPACE}_{i,\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon) \\ \subseteq \; & \mathrm{SPACE}_\Upsilon(O((f(n))^2 \cdot \log(f(n)))) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. Let $\Upsilon \in \mathrm{voc}_{\mathrm{finite}}$ be an order-solvent and graph-solvent vocabulary, f be a space constructible function and $i \in \{\mathrm{el}, \mathrm{rule}\}$.
Let $(L_1, L_2) \in \mathrm{NSPACE}_{i,\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon)$. So, there exists a non-deterministic ASM $\mathcal{A} = (P, L_2)$ of kind i such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{NSPACE}_i(P, J) \leq f(\mathrm{size}(J))$ for all $I \in L_2$.
Therefore, no run of P on any $I \in L_2$ contains a structure of size $> f(\mathrm{size}(I))$ resp. there is no structure reachable from I by use of P such that it has a size $> f(\mathrm{size}(I))$. I.e., all structures reachable from I have size $\leq f(\mathrm{size}(I))$.
Consider all Υ-structures J of size $\leq f(\mathrm{size}(I))$ (such that they belong to the considered case). So, $|\mathrm{Living}(J)| \leq f(\mathrm{size}(I))$.
Define $x := f(\mathrm{size}(I))$. Let $g \in \Upsilon$. Then, for such a structure and a given (living) universe, there are are at most x^x possiblities to interpret g. The reason for this is, that $|\mathrm{Living}(I)| \leq x$ and because of the definition of the size of a structure, the arity of g does not need to be considered here.

So, there is are $\leq x \cdot (x^x)^{|\Upsilon|} = x \cdot (2^{x \cdot \log x})^{|\Upsilon|} = x \cdot (2^{|\Upsilon| \cdot x \cdot \log x})$.
Define the set of all states J reachable from I by use of P as over Υ with States(PI) (where the difference caused by importing different elements from the reserve is not considered).
So, the following holds: $I \in L_1$ iff there exists an accepting state $J \in \text{States}(P, I)$.
If there is an accepting run of P on I then there is also a run of length $\leq x \cdot (2^{|\Upsilon| \cdot x \cdot \log x}) = (2^{(|\Upsilon| \cdot x+1) \cdot \log x})$. So, I is accepted by P iff there is an accepting run of length $\leq x \cdot (2^{|\Upsilon| \cdot x \cdot \log x})$ of P on I.
Now, define a deterministic procedure Reach with output: $\text{Reach}(J_1, J_2, k) = \text{true}$ iff J_2 can be reached from J_1 within 2^k macrosteps by use of P.
The procedure will not be given as a pure ASM-program, the copying etc. will be left out because this would only decrease the comprehensibility. It is partly described in pseudo-Pascal code and instead of E, $\vdash_P$ is used:

```
Reach(C_1, C_2, k)
if k = 0 then
  if [(C_1 = C_2) ∨ (C_1 ⊢_P C_2)]
  then
    block
      Output := true
      Mode := false
    endblock
  else
    block
      Output := false
      Mode := false
    endblock
  endif
endif
if k > 0 then
  for all C ∈ States(f, I) do
    if [Reach(C_1, C, k − 1) = true ∧ Reach(C, C_2, k − 1) = true]
    then
      block
        Output := true
        Mode := false
      endblock
    endif
endif
```

"for all" has to be read as a sequential instruction here. It means to go sequentially through all possibilities.
Define $s(n, k)$ as the maximal space to calculate $\text{Reach}(C_1, C_2, k)$ for $C_1, C_2 \in$

$\text{States}(f, I)$ where $n = \text{size}(I)$. Then the following holds:

- $s(n, 0) \leq 2 \cdot f(n)$
- $s(n, k+1) \leq c \cdot f(n) + s(n, k)$ for a suitable constant c.

From this one gets that $s(n, k) \leq (c \cdot k \cdot f(n) + 2) \cdot f(n)$ So, $L = \text{Acc}(P)$ can be decided by the following deterministic program (again given in pseudo-Pascal-code):

```
Input:  I
for all J ∈ States(f, I), J is an accepting state do
  if Reach(I, J, (|Υ| · x + 1) · log x)) = true
  then accept, end
od
reject
```

The space needed by this algorithm is:

$$\begin{aligned} &\leq O(f(\text{size}(I))) + s(\text{size}(I), (|\Upsilon| \cdot f(\text{size}(I) + 1) \cdot \log(f(\text{size}(I))) \\ &\leq O((f(\text{size}(I)))^2 \cdot \log(f(\text{size}(I)))) \end{aligned}$$

□

Lemma 5.16. *Let $f : \mathbb{N} \to \mathbb{N}$. Then* $\text{TIME}^{\text{micro}}(f(n)) \subseteq \text{SPACE}(f(n))$.

Proof. Because per one microstep the structure size can just grow by one, this lemma is trivial. □

Because per one macrostep the structure size can just grow by c, the following lemma is trivial.

Lemma 5.17. *Let $f : \mathbb{N} \to \mathbb{N}$, $c \in \mathbb{N}$. Then*
$\{L$: *there exists* $\Upsilon \in \text{voc}$ *such that* L *is a set of* Υ*-structures and there exists a* c*-limited ASM-program* P *such that* $L = \text{Acc}(\mathcal{A})$ *and* $\text{TIME}^{\text{macro}}(P, I) \leq f(\text{size}(I))$ *for every* Υ*-structure* $I\} \subseteq \text{SPACE}(c \cdot f(n))$

Theorem 5.18. *Let* $\Upsilon \in \text{voc}_{\text{finite}}$ *be an order-solvent and graph-solvent, and* $f : \mathbb{N} \to \mathbb{N}$.
Then

$$\begin{aligned} &\text{SPACE}_\Upsilon(f(n)) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \\ \subseteq\ &\text{TIME}_\Upsilon^{\text{macro}}(2^{O(f(n) \cdot \log(n))}) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. Let $\Upsilon \in \text{voc}_{\text{finite}}$ be an order-solvent and graph-solvent vocabulary and f be a macro time constructible function.
Let $(L_1, L_2) \in \text{SPACE}_\Upsilon(f(n)) \restriction (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon)$. So, there exists

a (deterministic) ASM $\mathcal{A} = (P, L_2)$ such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{SPACE}(P, J) \leq f(\mathrm{size}(J))$ for all $I \in L_2$.
Let $I \in L_2$. Consider all Υ-structures J of size $\leq f(\mathrm{size}(I))$ (such that they belong to the considered case). So, $|\mathrm{Living}(J)| \leq f(\mathrm{size}(I))$.
Define $x := f(\mathrm{size}(I))$. Let $g \in \Upsilon$. Then, for such a structure and a given (living) universe, there are are at most x^x possiblities to interpret g. The reason for this is, that $|\mathrm{Living}(I)| \leq x$ and because of the definition of the size of a structure, the arity of g does not need to be considered here.
So, there is are $\leq x \cdot (x^x)^{|\Upsilon|} = x \cdot (2^{x \cdot \log x})^{|\Upsilon|} = x \cdot (2^{|\Upsilon| \cdot x \cdot \log x})$.
Therefore, the number of Υ-structures reachable from I
by use P is $\leq 2^{O(f(\mathrm{size}(I)) \cdot \log(\mathrm{size}(I)))}$.
With the usual algorithm working on the computation graph, this theorem can be concluded. □

5.5 Completeness

Definition 5.19. Let PL be the set of Boolean expressions (propositional logic). Then QPL is the smallest set fulfilling the following conditions:

- $\mathrm{PL} \subseteq \mathrm{QPL}$
- $\psi, \varphi \in \mathrm{QPL} \Longrightarrow (\psi \vee \varphi), (\psi \wedge \varphi), \neg\psi \in \mathrm{QPL}$
- $\psi \in \mathrm{QPL}$, let X be a propositional variable $\Longrightarrow \exists X \psi, \forall X \psi \in \mathrm{QPL}$

Definition 5.20. Define $\mathrm{QBF} := (L_1^{\mathrm{QBF}}, L_2^{\mathrm{QBF}})$ where
$L_1^{\mathrm{QBF}} := \{\psi \in \mathrm{QPL} : \psi$ does not contain any free variable$\}$ and $L_2^{\mathrm{QBF}} := \{\psi \in \mathrm{QPL} : \psi$ does not contain any free variable, ψ is satisfiable$\}$

Lemma 5.21. $\mathrm{QBF} \in \mathrm{PSPACE}_{\mathrm{finite}}$.

Proof. Assume that no variable is reused (else rename them). Then leave away the quantifiers and try all interpretations in the lexicographical order. Only the actual interpretation has to be saved. Accept according to the quantifiers. □

Theorem 5.22. *Let $\Upsilon \in \mathrm{voc}_{\mathrm{finite}}$ be an order-solvent and copy-solvent vocabulary.*
QBF *is* $\mathrm{PSPACE}_\Upsilon \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyCopy}_\Upsilon)$*-hard via polynomial time reduction.*

Proof. Let $(L_1, L_2) \in \mathrm{PSPACE}_\Upsilon \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyCopy}_\Upsilon)$. So, there exists a (deterministic) ASM $\mathcal{A} = (Q, L_2)$ and a polynomial p such that $\mathrm{SPACE}(Q, I) \leq p(\mathrm{size}(I))$ for all $I \in L_2$ and $\mathrm{Acc}(Q) = L_1$. W.l.o.g., one can assume that Q has the normal form of lemma 2.5.
Let $N_{\mathcal{A},I} := \{w_i : 1 \leq i \leq |\Upsilon_{\mathrm{dyn}}| \cdot \max_{g \in \Upsilon_{\mathrm{dyn}}}(\mathrm{arity}(g)+1) \cdot p(\mathrm{size}(I)) - |\mathrm{Active}(I)|\} \cup$

Active(I) where $\{v_i : 1 \leq i \leq |\Upsilon_{\text{dyn}}| \cdot \max_{g\in\Upsilon_{\text{dyn}}}(\text{arity}(g)+1) \cdot p(\text{size}(I)) - |\text{Active}(I)|\} \cap \text{Active}(I) = \emptyset$. Therefore, $|N_{\mathcal{A},I}| = |\Upsilon_{\text{dyn}}| \cdot \max_{g\in\Upsilon_{\text{dyn}}}(\text{arity}(g)+1) \cdot p(\text{size}(I))$ and because of lemma 3.2, no state in a run of Q on I has more than $|N_{\mathcal{A},I}|$ active elements. The elements in $N_{\mathcal{A},I} - \text{Active}(I)$ represent the elements imported durring the run or to be more precise it is a superset.
A state can be characterized via the contents of its locations. Introduce propositional variables where $f \in \Upsilon_{\text{dyn}}$ is an r-ary function name, $a_1, ..., a_r, b \in N_{\mathcal{A},I}$. $X_{f,a_1,...,a_r,b,t}$ is true iff the content of the location $(f, a_1, ..., a_r)$ is b at time t.
The Boolean expression uses the following components:

State($\bar{X}$) : $\bar{X}$ characterizes a configuration, i.e., it assigns exactly one "value" to each term occuring in Q.
$\text{State}(\bar{X}) = \bigwedge_{f\in\Upsilon,\text{arity}(f)=r} \bigwedge_{a_1,...,a_r,b\in N_{\mathcal{A},I}} (X_{f,a_1,...,a_r,b} \to \bigwedge_{a\in N_{\mathcal{A},I},a\neq b} \neg X_{f,a_1,...,a_r,a})$

Input$_I(\bar{X})$: $\bar{X}$ characterizes the input of Q. More precisely, $\text{Input}_I(\bar{X})$ has the following form: $\text{Input}_I(\bar{X})$ is the conjunction of all $X_{f,a_1,...,a_r,b}$ such that $f^I(a_1, ..., a_r) = b$, $f \in \Upsilon$, $r = \text{arity}(f)$, $a_1, ..., a_r, b \in N_{\mathcal{A},I}, i \in \{1, ..., k\}\}$. If $a_j \in N_{\mathcal{A},I} - \text{Active}(I)$ for an $j \in \{1, ..., r\}$ then $f^I(a_1, ..., a_r) = \text{undef}$.

Accept($\bar{X}$) : $\bar{X}$ characterizes an accepting state. $\text{Accept}(\bar{X}) = \text{State}(\bar{X}) \wedge X_{\text{Mode,false}} \wedge X_{\text{Output,true}}$

Next($\bar{X}, \bar{Y}$) : $\bar{Y}$ is the successor of $\bar{X}$. Because Q is deterministic and in the normal form of lemma 2.5, this expression has only to describe the updates which are each placed inside an if-clause `if` g `then` $s_0 := t_0$.
First, define $\text{Eval}(s, b, \bar{X})$ for a term s constructed by use of function names from Υ and elements from $\{v_i : i \in \{1, ..., k\}\}$, $b \in N_{\mathcal{A},I}$. $\text{Eval}(s, b, t)$ is defined by induction on s:

- if $s = c$ for a nullary function name $c \in \Upsilon$ or then $\text{Eval}(s, b, \bar{X}) = X_{s,b}$.
- if $s = v_i$ for an $i \in \{1, ..., k\}$. Then $\text{Eval}(s, \bar{X})$ is the i-th $w \in N_{\mathcal{A},I} - \text{Active}(I)$ (the order is given by the indices) such that $X_{f,a_1,...,a_r,\text{undef}} = \text{true}$ for all $f \in \Upsilon$ and $a_1, ..., a_r \in N_{\mathcal{A},I}$ such that at least one of the a_j ($j \in \{1, ..., r\}$) is equal to w.
- if $s = f(s_1, ..., s_r)$ for $f \in \Upsilon$, $\text{arity}(f) = r$ and s_1, ..., s_r are terms constructed by use of function names from Υ and elements from $N_{\mathcal{A},I} - \text{Active}(I)$ then
$\text{Eval}(s, b, \bar{X}) = \bigvee_{b_1,...,b_r\in N_{\mathcal{A},I}} \text{Eval}(s_1, b_1, \bar{X}) \wedge ... \wedge \text{Eval}(s_r, b_r, \bar{X}) \wedge X_{f,b_1,...,b_r,b}$

Consider the block of if-clauses inside the program Q. For every $M \in \mathcal{P}(\{1, ..., n\})$ define a Boolean expression $\psi_{M,\bar{X},\bar{Y}}$: For every if-clause `if` c `then` $f(u_1, ..., u_{\text{arity}(f)}) := g(v_1, ..., v_{\text{arity}(g)})$ introduce the following:

$$(\bigwedge_{i\in M} \text{translation}(c_i, \bar{X}) \wedge \bigwedge_{i\notin M} \neg\text{translation}(c_i, \bar{X})) \to \varphi_{M,\bar{X},\bar{Y}}$$

$\varphi_{M,\bar{X},\bar{Y}}$ describes the updates caused by those if-clauses i such that $i \in M$. Other changes of the contents of the locations are avoided. The updates can be grouped by the function names given in the updated locations. Let $M_f := \{f(s_1^1, ..., s_r^1) := s_0^1, ..., f(s_1^r, ..., s_r^r) := s_0^r\}$ be the set of all updates which update a location location with function name f and belong to the i-th if-clause for an $i \in M$. So, $\varphi_{M,\bar{X},\bar{Y}} = \bigwedge_{f \in \Upsilon} \varphi_{M,f,\bar{X},\bar{Y}}$. In $\varphi_{M,f,\bar{X},\bar{Y}}$ only those updates are considered which deal with f. Now, $\varphi_{M,f,\bar{X},\bar{Y}}$ will be given:

$$
\begin{aligned}
&\varphi_{M,f,\bar{X},\bar{Y}} = \\
&\qquad \bigwedge_{f(s_1,...,s_r):=s_0 \in M_f} \ \bigwedge_{b \in N_{\mathcal{A},I}} \mathrm{Eval}(s_0, b, \bar{X}) \to \mathrm{Eval}(f(s_1, ..., s_r), b, \bar{Y}) \\
&\wedge \quad \bigwedge_{b_1,...,b_r \in N_{\mathcal{A},I}} \Big(\bigwedge_{f(t_1,...,t_r):=s \in M_f} \Big(\neg \bigwedge_{i \in \{1,...,r\}} \mathrm{Eval}(s_i, b_i, \bar{X})\Big)\Big) \\
&\qquad \to \Big(\bigwedge_{b \in N_{\mathcal{A},I}} X_{f,b_1,...,b_r,b} \to Y_{f,b_1,...,b_r,b}\Big)
\end{aligned}
$$

translation$(c, \bar{X})$ is the translation of the guard c in the state given by $\bar{X}$. It is defined inductively as follows:

- if c is an equality $s_1 = s_2$ where s_1 and s_2 are terms then translation$(c, \bar{X})$ = Eval$(s_1, \bar{X}) \leftrightarrow$ Eval$(s_2, \bar{X})$.
- if c is a negation $\neg c'$ then translation$(c, \bar{X}) = \neg$translation$(c', \bar{X})$
- if c is a conjunction $c_1 \wedge c_2$ then translation$(c, \bar{X})$ = translation$(c_1, \bar{X}) \wedge$ translation$(c_2, \bar{X})$
- if c is a disjunction $c_1 \vee c_2$ then translation$(c, \bar{X})$ = translation$(c_1, \bar{X}) \vee$ translation$(c_2, \bar{X})$

These formulae can be constructed from I (and Q) in polynomial time (compared to the size of I). Furthermore, define the following:

Eq$(\bar{X}, \bar{Y})$**:** $\bar{X}$ and $\bar{Y}$ describe the same state.
$\mathrm{Eq}(\bar{X}, \bar{Y}) = \bigwedge_{f \in \Upsilon} \bigwedge_{a_1,...,a_{\mathrm{arity}(f)}, b \in N_{\mathcal{A},I}} X_{f,a_1,...,a_{\mathrm{arity}(f)},b} \leftrightarrow Y_{f,a_1,...,a_{\mathrm{arity}(f)},b}$

Now, a formula Reach$_m(\bar{X}, \bar{Y})$ is constructed which is interpreted as follows: $\bar{X}$ and $\bar{Y}$ characterize states and the state characterized by $\bar{Y}$ is reachable from the state characterized by $\bar{X}$ within at most 2^m macrosteps (by use of Q). For $m = 0$, define:
$\mathrm{Reach}_0(\bar{X}, \bar{Y}) := \mathrm{State}(\bar{X}) \wedge \mathrm{State}(\bar{Y}) \wedge (\mathrm{Eq}(\bar{X}, \bar{Y}) \vee \mathrm{Next}(\bar{X}, \bar{Y}))$

Inductively, define $\mathrm{Reach}_{m+1}(\bar{X},\bar{Y})$:

$$\begin{aligned}\mathrm{Reach}_{m+1}(\bar{X},\bar{Y}) := \exists\bar{Z}\forall\bar{U}\forall\bar{V}\quad & (((\mathrm{Eq}(\bar{X},\bar{U}) \wedge \mathrm{Eq}(\bar{Z},\bar{V})) \vee (\mathrm{Eq}(\bar{Z},\bar{U}) \wedge \mathrm{Eq}(\bar{Y},\bar{V}))) \\ & \to \mathrm{Reach}_m(\bar{U},\bar{V}))\end{aligned}$$

The following holds:
There exists a polynomial q such that that

- $|\mathrm{Reach}_0(\bar{X},\bar{Y})| \leq q(\mathrm{size}(I))$.
- $|\mathrm{Reach}_{m+1}(\bar{X},\bar{Y})| \leq |\mathrm{Reach}_m(\bar{X},\bar{Y})| + q(\mathrm{size}(I))$

Therefore $|\mathrm{Reach}_m(\bar{X},\bar{Y})| = O(m \cdot q(\mathrm{size}(I)))$
If Q accepts I within time $p(\mathrm{size}(I))$ then it also accepted within $\leq 2^{c \cdot p(\mathrm{size}(I)) \cdot \log(p(\mathrm{size}(I)))}$ macrosteps for a suitable constant c. Let $m := c \cdot p(\mathrm{size}(I)) \cdot \log(p(\mathrm{size}(I)))$ and $\psi_{\mathcal{A},I} := \exists\bar{X}\exists\bar{Y}(\mathrm{Input}_I(\bar{X}) \wedge \mathrm{Accept}(\bar{X}) \wedge \mathrm{Reach}_m(\bar{X},\bar{Y}))$
Obviously, $\psi_{\mathcal{A},I}$ can be constructed within polynomial time and the following holds: $\psi_I \in \mathrm{QBF}$ iff $I \in \mathrm{Acc}(Q)$. $\square$

Chapter 6

Alternating Abstract State Machines

The definitions in this chapter make sense for both kinds of non-determinism. Therefore, the parameter $i \in \{\text{el}, \text{rule}\}$ is used again.

6.1 The Basic Model

Definition 6.1. Let $\mathcal{A} = (P, \text{Input})$ be a non-determinstic ASM and $I \in \text{Input}$. The *computation tree* $\mathcal{T}_{\mathcal{A},I}$ is a tree $T = (V, E)$ together with a labeling function L such that:

- L maps the root of T to I.
- $A \vdash_P B$ iff for every node $u \in V$ with $L(u) = A$ there exists exactly one child v of u such that $L(v) = B$.
- If $A \vdash_P B$ is not true and $u \in V$ with $L(u) = A$ then $L(v) \neq B$ for all children v of u.

Note that the nodes of the computation tree are not the states the ASM. This is an important difference compared to the definition of a computation graph. The computation tree is the winding off of the computation graph. The computation tree is not necessarily a tree because a state might be reachable on different ways. Furthermore, note that, for element-non-determinism, there is no bound for the branching degree in general. It might be infinite and there are even cases where a node has uncountable many children (e.g., with $(\mathbb{R}, +, \cdot, 0, 1)$ as background structure). This is the reason why most results for alternating ASMs (which are defined in the following) hold only in the special case which is considered in a later chapter.
P accepts I iff there is a path in $\mathcal{T}_{\mathcal{A},I}$ from the root to a node labeled with an accepting state.

Another way to define acceptance is the following one: Every state of the ASM $\mathcal{A} = (P, I)$ defines a subtree T_J (up to isomorphism) whose root is labeled by J. T_J is accepting if

- J is an accepting state or
- there exists a state K labeling a child of the root such that T_K is accepting.

P accepts $I \in$ Input iff $\mathcal{T}_{A,I}$ is accepting.

Definition 6.2. An *alternating ASM* is a element-non-deterministic or rule-non-deterministic ASM over a vocabulary $\Upsilon \in$ voc which contains a Boolean, nullary function $\forall$.

So, the states of an alternating ASM are divided into existential (enabling resp. disabling), universal (enabling resp. disabling), final (accepting resp. rejecting). A state is

- existential iff Mode = true and $\forall$ = false
- universal iff Mode = true and $\forall$ = true
- final iff Mode = false

Accepting, rejecting, disabling, enabling are defined as before.
The computation tree $\mathcal{T}_{\mathcal{A},I}$ for an alternating ASM $\mathcal{A} = (P, \text{Input})$ and a state $I \in$ Input is defined as for a non-deterministic ASM. The definiton of accepting is different: T_J is accepting if

- J is an accepting state or
- J is an existential state and there is a state K with $J \vdash_P K$ such that T_K is an accepting subtree or
- J is a universal state and for all states K with $J \vdash_P K$ it is true that T_K is an accepting subtree.

P accepts $I \in$ Input iff $T_I = \mathcal{T}_{A,I}$ is accepting.
Let $\mathcal{A} = (P, \text{Input})$ be an alternating ASM (either element-non-deterministic or rule-non-deterministic). Then $\text{Acc}(\mathcal{A}) := \{I : I \in \text{Input and } \mathcal{T}_{P,I} \text{ is accepting}\}$
Another possibility to define acceptance is given by a game on the computation tree. Two players, $\forall$ and $\exists$, take part in the game. The rules are the following: The game starts at the root of $\mathcal{T}_{\mathcal{A},I}$. At every node labeled by an existential state, $\exists$ chooses a successor. At every node labeled by a universal state, $\forall$ chooses a successor. When a node labeled by an accepting state is reached then $\exists$ wins. When a node labeled by a rejecting state is reached then $\forall$ wins. Obviously, P accepts I iff $\exists$ has a winning strategy for the game on $\mathcal{T}_{A,I}$.

6.2 Alternating Complexity Classes

Definition 6.3. Let $i \in \{\text{el}, \text{rule}\}$, $f : \mathbb{N} \to \mathbb{N}$. Define the following complexity classes:

1. Let P be a program over $\Upsilon \in voc$ and $I \in \text{Inputstates}(P)$. Then $\text{ATIME}_i^{macro}(P, I)$ is the length longest path in $\mathcal{T}_{P,I}$. $\text{ATIME}_i^{macro}(P, I)$ is maximal number of microsteps executed on a path in $\mathcal{T}_{P,I}$. $\text{ASPACE}_i(P, I)$ is the maximal size of a label in $\mathcal{T}_{P,I}$.

2. $\text{ATIME}_{i,\Upsilon}^{\text{macro}}(f(n)) := \{(L_1, L_2) : L_1, L_2$ are sets of Υ-structures of finite size and there exists an alternating ASM $\mathcal{A} = (P, L_2)$ (kind i) such that $L_1 = \text{Acc}(\mathcal{A})$ and the longest path in $\mathcal{T}_{\mathcal{A},I}$ has length $\leq f(\text{size}(I))$ for all $I \in L_2\}$

3. $\text{ATIME}_i^{\text{macro}}(f(n)) := \bigcup_{\Upsilon \in \text{voc}} \text{ATIME}_{i,\Upsilon}^{\text{macro}}(f(n))$

4. $\text{ATIME}_{i,\Upsilon}^{\text{micro}}(f(n)) := \{(L_1, L_2) : L_1, L_2$ are sets of Υ-structures of finite size and there exists an alternating ASM $\mathcal{A} = (P, L_2)$ (kind i) such that $L_1 = \text{Acc}(\mathcal{A})$ and on every path in $\mathcal{T}_{\mathcal{A},I} \leq f(\text{size}(I))$ microsteps are done for every $I \in L_2\}$

5. $\text{ATIME}_i^{\text{micro}}(f(n)) := \bigcup_{\Upsilon \in \text{voc}} \text{ATIME}_{i,\Upsilon}^{\text{macro}}(f(n))$

6. $\text{ASPACE}_{i,\Upsilon}(f(n)) := \{(L_1, L_2) : L_1, L_2$ are sets of Υ-structures of finite size and there exists an alternating ASM $\mathcal{A} = (P, L_2)$ of kind i such that $L_1 = \text{Acc}(\mathcal{A})$ and for all labels J in $\mathcal{T}_{\mathcal{A},I}$ $\text{size}(J) \leq f(\text{size}(I))$ for all $I \in L_2\}$

7. $\text{ASPACE}_i(f(n)) := \bigcup_{\Upsilon \in \text{voc}} \text{ASPACE}_{i,\Upsilon}(f(n))$

Theorem 6.4. *Let $\Upsilon \in$ voc be an order-solvent and graph-solvent vocabulary, f be a space constructible function and $f(n) \geq n$.*
Then

$$\begin{aligned} & \text{ATIME}_{\text{rule},\Upsilon}^{\text{macro}}(f(n)) \upharpoonright (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \\ \subseteq \; & \text{SPACE}_\Upsilon((f(n)+1) \cdot f(n)) \upharpoonright (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. Let $\Upsilon \in$ voc be an order-solvent and graph-solvent vocabulary and f be a space constructible function.
Let $(L_1, L_2) \in \text{NTIME}_{\text{rule},\Upsilon}^{\text{macro}}(f(n)) \upharpoonright (\text{ActiveOrder}_\Upsilon \cap \text{EmptyGraph}_\Upsilon)$. So, there exists an alternating ASM (rule-non-determinism) $\mathcal{A} = (P, L_2)$ such that $L_1 = \text{Acc}(\mathcal{A})$ and $\text{ATIME}_{\text{rule}}^{\text{macro}}(P, J) \leq f(\text{size}(J))$ for all $I \in L_2$.
There exists $r \in \mathbb{N}$ such that for every structure J, which is reachable from an element of L_2, $|\{K : J \vdash_P K\}| \leq r$. There is a bound for r given by P:

$r \leq 2^{\text{number of guess-rule rules in } P}$ (this is obvious).
An algorithm similar to the one in the proof of theorem 4.35 but with the following differences:

- Instead of the computation graph, the computation tree is constructed.
- Do not always accept if an accepting state is reached. The acceptance depends on the content of $\forall$, i.e. whether the considered state is an existential or universal state. If it is existential then it is enough that one state is reachable which is accepting. Else all successors have to be accepting.
- It is not necessary to construct the complete tree at once. Only the actual path starting at the root has to be saved. The reason is that repititions does not need to be avoided because every state might occur more than once in the tree. The only thing that has to be saved is which is the next update.

This algorithm computes for every state J in $\mathcal{T}_{\mathcal{A},I}$ whether the subtree T_J is accepting (so, Output = true) or rejecting (so, Output = false).
Obviously, this program is correct. It decides whether I is accepted by P. Q is a deterministic program such that $\mathrm{Acc}(Q) = L$.
Now, consider the space needed by Q. Let J be a vertex in $\mathcal{T}_{\mathcal{A},I}$ of height t, i.e., all computations starting at J need at most t steps. Then the following holds:

- $\mathrm{SPACE}(J, P) = \mathrm{size}(J)$ if $t = 0$
- $\mathrm{SPACE}(J, P) = \max_{J \vdash_P K}(\mathrm{size}(K) + \mathrm{SPACE}(J, P)$ if $t > 0$

Because K with $J \vdash_P K$ has height $t - 1$, one gets:

$$\mathrm{SPACE}(J, P) \leq (t + 1) f(\mathrm{size}(J))$$

and as a consequence:

$$\mathrm{SPACE}(I, P) \leq (f(\mathrm{size}(I)) + 1) \cdot f(\mathrm{size}(I))$$

□

The following corollary is an obvious consequence of the preceding theorem.

Corollary 6.5. *Let $\Upsilon \in \mathrm{voc}$ be an order-solvent and copy-solvent vocabulary, f be a space constructible function and $f(n) \geq n$.*
Then
$\mathrm{ATIME}^{\mathrm{micro}}_{\mathrm{rule},\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyCopy}_\Upsilon) \subseteq \mathrm{SPACE}_\Upsilon((f(n) + 1) \cdot f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyCopy}_\Upsilon)$.

Theorem 6.6. *Let $\Upsilon \in \mathrm{voc}_{\mathrm{finite}}$ be an order-solvent and graph-solvent vocabulary, f be a space constructible function and $f(n) \geq n$.*
Then

$$\begin{aligned} & \mathrm{ATIME}^{\mathrm{macro}}_{\mathrm{el},\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon) \\ \subseteq \; & \mathrm{SPACE}_\Upsilon((f(n)+1) \cdot f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. The proof of this theorem can be done analogously to the proof of theorem 6.4. The only difference is the bound on the number of possible successors. Instead of $2^{\text{number of guess-rule rules in } P}$ the bound is $(|\mathrm{Active}(I)|+2)^{\text{number of guess-element rules in} P}$
The algorithm is the same and the needed space is calculated in the same way. □

Theorem 6.7. *Let $\Upsilon \in \mathrm{voc}$ be an order-solvent and graph-solvent vocabulary, f be a space constructible function and $f(n) \geq n$.*
Then

$$\begin{aligned} & \mathrm{ASPACE}_{\mathrm{rule},\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon) \\ \subseteq \; & \mathrm{TIME}^{\mathrm{micro}}_\Upsilon(2^{O(n)}) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon) \end{aligned}$$

Proof. Let $\Upsilon \in \mathrm{voc}$ be an order-solvent and copy-solvent vocabulary and f be a space constructible function.
Let $(L_1, L_2) \in \mathrm{ASPACE}_{\mathrm{rule},\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyCopy}_\Upsilon)$. So, there exists an alternating ASM (rule-non-determinism) $\mathcal{A} = (P, L_2)$ such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{ASPACE}_{\mathrm{rule}}(P, J) \leq f(\mathrm{size}(J))$ for all $I \in L_2$ and therefore, for every $I \in L_2$ $\mathrm{size}(K) \leq f(\mathrm{size}(I))$ for every label K in $\mathcal{T}_{\mathcal{A},I}$.
So, there exists a constant d such that there are at most $2^{d \cdot n}$ structures of size $\leq f(n)$ because: Per one guess-rule rule there are two possibilities to continue. So, for every structure J there are at most $2^{\text{number of guess-rule rules in } P}$ structures K such that $J \vdash_P K$. With $d :=$ number of guess-rule rules in P one gets the proposition from above.
Continuing as in the case of non-determinism using the computation tree instead of the computation graph and with the changes already described in the proof of theorem 6.4 proves this theorem. □

Theorem 6.8. *Let $f(n) \geq n$ and $i \in \{\mathrm{el}, \mathrm{rule}\}$. Then* $\mathrm{ASPACE}_{i,\mathrm{finite}}(f(n)) \subseteq \bigcup_{c>0} \mathrm{TIME}^{\mathrm{macro}}_{i,\mathrm{finite}}(2^{c \cdot n \cdot \log n})$

Proof. Let $(L_1, L_2) \in \mathrm{ASPACE}_{i,\mathrm{finite}}(f(n))$ So, there exists an alternating ASM (i-th kind) $\mathcal{A} = (P, L_2)$ such that $\mathrm{Acc}(P) = L_1$ and for every structure $I \in L_2$ it is true that $\mathrm{size}(K) \leq f(\mathrm{size}(I))$ for every label in $\mathcal{T}_{I,P}$.
So, there exists a constant d such that there are at most $2^{d \cdot n \cdot \log n}$ structures of size $\leq f(n)$ because: Let J be a structure of size $f(n)$. So, there are $\leq f(n)$

active elements in the universe of J. Let Υ be the vocabulary of J. For $g \in \Upsilon$ there are $(f(n))^{f(n)}$ possibilities to interprete it over a given universe of size $f(n)$. So, given a universe of size $f(n)$, there are $(f(n))^{|\Upsilon|\cdot f(n)}$ possibilities to build an Υ-structure over this universe.

Continuing as in the case of non-determinism using the computation tree instead of the computation graph and with the changes already described in the proof of theorem 6.4 proves this theorem. □

Theorem 6.9. *Let $\Upsilon \in \mathrm{voc}_{\mathrm{finite}}$ be an order-solvent and graph-solvent vocabulary, f be a space constructible function and $f(n) \geq n$.*
Then

$$\begin{array}{ll} & \mathrm{NSPACE}_{\mathrm{el},\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon) \\ \subseteq & \mathrm{ATIME}^{\mathrm{macro}}_{\mathrm{el},\Upsilon}(f^2(n) \cdot \log f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon) \end{array}$$

Proof. Let $\Upsilon \in \mathrm{voc}_{\mathrm{finite}}$ be an order-solvent and graph-solvent vocabulary and f be a space constructible function.
Let $(L_1, L_2) \in \mathrm{NTIME}^{\mathrm{macro}}_{\mathrm{rule},\Upsilon}(f(n)) \restriction (\mathrm{ActiveOrder}_\Upsilon \cap \mathrm{EmptyGraph}_\Upsilon)$. So, there exists an alternating deterministic ASM (element-non-determinism) $\mathcal{A} = (P, L_2)$ such that $L_1 = \mathrm{Acc}(\mathcal{A})$ and $\mathrm{NSPACE}_{\mathrm{rule}}(P, J) \leq f(\mathrm{size}(J))$ for all $I \in L_2$ and therefore, no run of P on any $I \in L_2$ with $\mathrm{size}(I) = n$ contains a state J with $\mathrm{size}(J) > f(n)$ and $\mathrm{Acc}(\mathcal{A}) = L_1$.
Consider all Υ-structures J of size $\leq f(\mathrm{size}(I))$ (such that they belong to the considered case). So, $|\mathrm{Living}(J)| \leq f(\mathrm{size}(I))$.
Define $x := f(\mathrm{size}(I))$. Let $g \in \Upsilon$. Then, for such a structure and a given (living) universe, there are are at most x^x possiblities to interpret g. The reason for this is, that $|\mathrm{Living}(I)| \leq x$ and because of the definition of the size of a structure, the arity of g does not need to be considered here.
So, there is are $\leq x \cdot (x^x)^{|\Upsilon|} = x \cdot (2^{x \cdot \log x})^{|\Upsilon|} = x \cdot (2^{|\Upsilon| \cdot x \cdot \log x})$.
Define the set of all states J reachable from I by use of P as over Υ with $\mathrm{States}(PI)$ (where the difference caused by importing different elements from the reserve is not considered).
One can construct a alternating ASM (element-non-determinsm) which works on the computation graph $\mathcal{T}_{\mathcal{A},I}$ and decides whether a state J_2 is reachable from a state J_1 within 2^t steps as follows:

- if $J_1 = J_2$ or $J_1 \vdash_P J_2$ (therefore, there is an edge from the vertex belonging to J_1 to the node belonging to J_2) then accept
- existentially guess a state J then universally choose to continue with checking whether J is reachable from J_1 within 2^{t-1} macrosteps or whether J_2 is reachable from J within 2^{t-1} macrosteps. This can be done by repeatedly applying the program.

The counting (i.e. t, $t-1$,...) can be done by using a structure from Order with t active elements and with each step delete one element.

The program is correct because C_2 is reachable from C_1 within $\leq 2^t$ (macro)steps iff there exists C such that C is reachable from C_1 within $\leq 2^{t-1}$ steps and C_2 is reachable from C within $\leq 2^{t-1}$ steps.
Let $g(t)$ the macro time maximally needed by the described ASM for a given t. Then the following holds:

- $g(0) = O(f(n))$
- for $t > 0$ one gets $f(t) = O(f(n)) + f(t-1)$

So, $g(t) = (t+1) \cdot O(f(n))$.
L can be decided as follows:
The onput is the input structure. Guess the accepting I_{acc} (on a copy of I). Accept iff $(I, I_{\text{acc}}, |\Upsilon| \cdot f(\text{size}(I)) \cdot \log f(\text{size}(I)))$ by Reach. The algorithm needs $(d \cdot c \cdot f(n) \cdot f(n) \cdot \log f(n) + 1) \cdot O(f(n)) = O(f^2(n) \cdot \log f(n))$ (alternating) macrosteps. $\square$

Theorem 6.10. *Let f be a space constructible function with $f(n) \geq n$, $\Upsilon \in$ $\text{voc}_{\text{finite}}$, $\vdash_P$ a non-static 2-ary function, last a non-static nullary function name. If $(L_1, L_2) \in \text{NSPACE}_{\text{el,finite},\Upsilon}(f(n))$ then there exists $c \in \mathbb{N}$ such that* $\text{Exp}((L_1, L_2), \vdash_P, \text{last}) \in \text{ATIME}^{\text{micro}}_{\text{el,finite},\Upsilon \cup \{\vdash_P, \text{last}\}}(c \cdot f^2(n) \cdot \log f(n))$

Proof. The proof can be done analogously to the one of the corresponding theorem for macrosteps. The constant c must be chosen such that P is c-micro-limited where P is an element-non-deterministic ASM-program which accepts L within space $f(n)$. $\square$

Chapter 7

Probabilistic Abstract State Machines

Only element-non-determinism is considered. The reason for this is given at the end of this chapter.
In this chapter, the notion of a probabilistic ASM (PASM) is introduced. The motivation is again given by the notion of a probabilistic Turing machine.
At the beginning of this chapter, one has the first problem is to find an adequate model for this. The first idea is the following one: A probabilistic ASM given by a program Q is an ASM whose input is a pair (I, X) where I is a an input structure for Q and X is a finite sequence of elements of the universe of I. I is the real input and X is a random sequence of r_Q-tuples of elements of the universe of I which directs the computation of the ASM. This idea has to be explained a little bit, especially the meaning of the sequence X. X is a random sequence of r_Q-tuples of elements of the living part of the universe of I. r_Q is the number of guess-element and guess-rule rules in the code of Q. A rule even counts if it is nested in an if-clause for example. One tuple describes the guesses of one macrostep of Q (if a guess-rule is not executed in a turn then the respective component is set to undef). So, a sequence of such tuples describes a run if the last tuple indicates a final state.
Let the number of macrosteps, a program Q does on an input (I, X), be limited by $p(\text{size}(I))$. So, w.l.o.g., one can assume that the length of X is $p(\text{size}(I))$. A σ-algebra can be defined as the $p(\text{size}(I))$-fold product of the r_Q-fold product of the potential set of the universe of I. The big problem with this definition arises when one wants to calculate the probability that Q accepts I. The components of an element of the σ-algebra are not stochastically independent from each other. There exists an injective map: run to set of sequence of tuples (there are sequences that do not describe any run or partial run of a given program). So, the probabilities in the i-th step depend on the steps (guesses in the steps) done before. One has to use conditional probabilities, because there might be guess-element rules where the set, which gives the condition, also depends on the elements chosen

before. Furthermore, this set depends on the Boolean-valued term, which is the guard of the considered guess-element rule (the Boolean-valued term defines a set of elements of the living part of the universe).
So, the calculation of the probabilities becomes very complicated for this definition and there are not many propositions which one can give about this. There are not enough assumptions to say something general.
The following definition will be the basis of this chapter.

Definition 7.1. A probabilistic ASM (PASM) $A = (Q, \text{Input})$ is a element-non-deterministic ASM whose input is a pair (I, X) where $I \in \text{Input}$ and X is a finite sequence of elements of the universe of I. I is the real input and X is a random sequence which directs the computation of the ASM.

With every structure, there has to be given a probability measure. The σ-algebra, on which this measure is defined, is the potential set of the universe of the structure.
A PASM $\mathcal{A} = (Q, \text{Input})$, where Q is a non-deterministic ASM-program Q, is called $p(n)$-microstep-limited if for every input (I, X), $I \in \text{Input}$, Q stops after at most $p(\text{size}(I))$ microsteps. W.l.o.g., one can assume that the length of X is $\leq p(\text{size}(I))$ and even $|X| = p(\text{size}(I))$ (if the length of X is smaller then add e.g. undef until the length is equal to $p(\text{size}(I))$. If a limit concerning the number of macrosteps is given, then a constant factor has to be introduced, namely, the number of microstep that can maximally be executed per one macrostep by the program.
Let $\mathcal{A} = (Q, \text{Input})$ be $p(n)$-microstep-limited. Then $\text{Acc}(\mathcal{A}) \subseteq \{(I, X) : I \in \text{Input}$ and X is a sequence of elements of the universe of I and $|X| \leq p(\text{size}(I))\}$. So, for I build the product σ-algebra $\mathcal{M} := \bigotimes_{i=1}^{p(\text{size}(I))}(\text{universe}(I))$. This is where X is taken from. Let P be the measure on $\mathcal{P}(\text{universe}(I)$ (as already mentioned above). Then there is an unambiguous probability measure $P^{p(\text{size}(I))}$ given by $P^{p(\text{size}(I))}(M_1 \times ... \times M_{p(\text{size}(I))}) = \Pi_{i=1}^{p(\text{size}(I))} P(M_i)$ for $M_j \in \mathcal{P}(\text{universe}(I)), j \in \{1, ..., p(\text{size}(I))\}$.
This is the right probability measure because one can assume the stochastic independence of the members of the sequence.
Now, for a structure $I \in \text{Input}$ the probability, that Q accepts I, can be defined as follows:

$$\Pr[\mathcal{A} \text{ accepts } I] = P^{p(\text{size}(I))}(\{X : (I, X) \in \text{Acc}(\mathcal{A})\})$$

For a probabilistic Turing machine, the input is a pair consisting of the real input and a finite random sequence of 0's and 1's. In the case of a polynomially bound probabilistic Turing machine, the length of the random sequence is polynomially in the size resp. the length of the real input. The members of the random sequence are stochastically independent and the probability for a 0 is the same as for a 1 equal to 0.5.

Lemma 7.2. *Let (L_1, L_2) be a pair of sets of Υ-structures of finite size. If there exists a polynomially bound PASM $\mathcal{A} = (Q, L_2)$ such that $L_1 = \{I : I \in L_2$ and $\Pr[Q$ accepts $I] > 0\}$ then $(L_1, L_2) \in \mathrm{NP}_{\mathrm{el}}$.*

Proof. Let $L_1 = \{I : I \in L_2 \text{ and } PR[Q \text{ accepts } I] > 0\}$ for a polynomially bound PASM $\mathcal{A} = (Q, L_2)$. So, for a suitable polynomial p (P is the given probability measure on the potential set of the universe of I), $P^{p(\mathrm{size}(I))}(\{X : (I, X) \in \mathrm{Acc}(\mathcal{A})\}) > 0$. So, for $B := \{(I, X) : I \in L_2$ and X is a sequence of length $p(\mathrm{size}(I))$ of elements of the universe of I and $(I, X) \in \mathrm{Acc}(\mathcal{A})\}$, $(B, L_2) \in$ P and therefore, for $A = \{I : I \in L_2 \text{ and } \exists X(|X| = p(\mathrm{size}(I))) \wedge (I, X) \in B\}$, $(A, L_2) in \mathrm{NP}_{\mathrm{el}}$. □

The other direction does not hold because there are probability measures where the probabilitiy 0 does not necessarily indicate impossibility.

Definition 7.3. Let (L_1, L_2) be a pair of sets of Υ-structures of finite size, $L_1 \subseteq L_2$.

- $(L_1, L_2) \in$ PP if there exists a polynomially bound PASM $\mathcal{A} = (Q, L_2)$ such that $L_1 = \{I : I \in L_2 \text{ and } \Pr[Q \text{ accepts } I] > 0.5\}$
- $(L_1, L_2) \in$ BPP if there exists a polynomially bound PASM $\mathcal{A} = (Q, L_2)$ such that
  - if $I \in L_1$ then $\Pr[Q \text{ accepts } I] \geq \frac{2}{3}$ and
  - if $I \in L_2 - L_1$ then $\Pr[Q \text{ accepts } I] \leq \frac{1}{3}$

  There are two kinds of error probabilities:
- positively false: $I \in L_2 - L_1$ but $\Pr[Q \text{ accepts } I] > 0$
- negatively false: $I \in L_1$ but $\Pr[Q \text{ accepts } I] < 1$, i.e., $\Pr[Q \text{ does not accept } I] = 1 - \Pr[Q \text{ accepts } I] > 0$

Definition 7.4. Let (L_1, L_2) be a pair of sets of Υ-structures of finite size, $L_1 \subseteq L_2$.

- $(L_1, L_2) \in$ RP if there exists a PASM $\mathcal{A} = (Q, L_2)$ such that
  - if $I \in L_1$ then $\Pr[Q \text{ accepts } I] \geq \frac{2}{3}$ and
  - if $I \in L_2 - L_1$ then $\Pr[Q \text{ accepts } I] = 0$ (the probability for positively false answers is equal to zero).
- $(L_1, L_2) \in \mathrm{Co} - \mathrm{RP}$ if there exists a PASM $\mathcal{A} = (Q, L_2)$ such that
  - if $I \in L_1$ then $\Pr[Q \text{ accepts } I] = 1$ (the probability for negatively false answers is equal to zero) and

  – if $I \in L_2 - L_1$ then $\Pr[Q \text{ accepts } I] \leq \frac{1}{3}$

- ZPP := RP ∩ Co − RP

Corollary 7.5. *The following inclusions hold:*

1. P ⊆ ZPP
2. ZPP ⊆ RP
3. ZPP ⊆ Co − RP
4. RP ⊆ BPP
5. Co − RP ⊆ BPP
6. BPP ⊆ PP

Proof. This corollary is a direct consequence from the definitions. □

The inclusion RP $\subseteq \mathrm{NP_{el}}$ does not hold because $\Pr[Q \text{ accepts } I] = 0$ does not (necessarily) imply that there is no accepting run of Q on I.
In the first moment, one could think that also the inclusion $\mathrm{NP_{el}} \subseteq \mathrm{PP}$ holds (as for probabilistic Turing machines). But this is not necessarily the case. The reason becomes clear when the assumptions of the following theorem are considered.

Theorem 7.6. *If $(L_1, L_2) \in \mathrm{NP_{el}}$ and for every $I \in L_1$ there exists a subset of the universe described by a Boolean expression (one expression for all) S_I such that the probability measure maps S_I to 0.5 and if there exists a polynomially bound PASM $\mathcal{A} = (Q, L_2)$ such that $L_1 = \{I : I \in L_2 \text{ and } \Pr[Q \text{ accepts } I] > 0\}$ then $(L_1, L_2) \in \mathrm{PP}$.*

Proof. Let $(L_1, L_2) \in \mathrm{NP_{el}}$ such that for every $I \in L_1$ there exists a subset of the universe defined by a Boolean expression (one expression for all) S_I such that the probability measure maps S_I to 0.5 and there exists a polynomially bound PASM Q such that $L_1 = \{I : I \in L_2 \text{ and } \Pr[Q \text{ accepts } I] > 0\}$.
Consider a PASM $\mathcal{A}' = (Q', L_2)$ which does the following for $I \in L_2$: (I, yX) is accepted iff either $y \in S_I$ or Q accepts (I, X).
Then the following holds:

$$\Pr[Q' \text{ accepts } I] = \frac{1}{2} + \frac{1}{2} \cdot \Pr[Q \text{ accepts } I]$$

So, $L_1 = \{I : I \in L_2 \text{ and } \Pr[Q' \text{ accepts } I] > \frac{1}{2}\} \in \mathrm{PP}$. □

Theorem 7.7. *For every $(L_1, L_2) \in \mathrm{BPP}$ and every polynomial $q(n)$ there exists a polynomially bound PASM $\tilde{\mathcal{A}} = (\tilde{Q}, L_2)$ which accepts L_1 with error probability $\leq 2^{-q(n)}$, i.e.,*

- *if* $I \in L_1$ *then* $\Pr[\tilde{Q}$ *accepts* $I] \geq 1 - 2^{q(\text{size}(I))}$ *and*
- *if* $I \in L_2 - L_1$ *then* $\Pr[\tilde{Q}$ *accepts* $I] \leq 2^{q(\text{size}(I))}$

Proof. Let $(L_1, L_2) \in$ BPP. So, there exist a polynomial $p(n)$ and a PASM $\mathcal{A} = (Q, L_2)$ which does at most $p(\text{size}(I))$ microsteps on any structure $I \in L_2$ and the error probability (concerning (L_1, L_2)) is $\leq \epsilon < \frac{1}{2}$.
Let (Q_k, L_2) be the PASM which accepts $(I, X_1...X_k)$, where $X_i, i \in \{1, ..., k\}$ is an $p(\text{size}(I))$-tuple of elements of the universe of I, iff $|\{i : (I, X_i) \in \text{Acc}(Q)\}| \geq \lceil \frac{k}{2} \rceil$. The number of micorsteps of Q_k is polynomially bound in the size of the input structure if $k = k(n)$ is polynomial in n.
Now, consider the error probability of Q_k. To calculate this, Chernoff's Theorem is used. It says the following:

> Let $Y_1, ..., Y_l$ be random variables with values from the set $\{0, 1\}$ such that $\Pr[Y_i = 1] = p$ and $\Pr[Y_i = 0] = 1 - p$ for an $0 \leq p \leq 1$, $i \in \{1, ..., k\}$. The sum $Y = \Sigma_{i=1}^{i=l} Y_i$ (the values are in $\mathbb{N}$) is a binomially distributed random variable. The expected value is $E(Y) = p \cdot l$. Let $d \geq 0$. Then the following holds:
>
> $$\Pr[Y - p \cdot k \geq d] \leq e^{-\frac{d^2}{4 \cdot k \cdot p \cdot (1-p)}} \leq e^{-\frac{d^2}{k}}$$
>
> and
>
> $$\Pr[p \cdot k - Y \geq d] \leq e^{-\frac{d^2}{k}}$$

Now, define for $X = X_1...X_k$ ($X_i, i \in \{1, ..., k\}$, as defined above) the following random variables:

- $Y_i(X) := 1$ if $(I, X_i) \in \text{Acc}(Q)$
- $Y_i(X) := 0$ else

With this, the following random variable can be defined: $Y := \Sigma_{i=1}^{k} Y_i$.
Then the following holds:

1. If $I \in L_2 - L_1$ then $p := \Pr[Y_i = 1] \leq \epsilon$ and

$$\Pr[Q_k \text{ accepts } I] = \Pr[Y \geq \frac{k}{2}]$$

According to Chernoff's theorem, one gets the following:

$$\Pr[Y \geq \frac{k}{2}] = \Pr[Y - p \cdot k \geq \frac{k}{2} - p \cdot k] \leq e^{-(\frac{1}{2} - p)^2 \cdot k} = 2^{-\Omega(k)}$$

2. If $I \in L_1$ then $\Pr[Q \text{ accepts } I] = \Pr[X_i = 1] = p \geq 1 - \epsilon$ and

$$\Pr[Q_k \text{ does not accept } I] = \Pr[Y < \frac{k}{2}]$$

According to Chernoff's theorem, one gets the following:

$$\Pr[Y < \frac{k}{2}] = \Pr[p \cdot k - Y \geq p \cdot k - \frac{k}{2}] \leq e^{-(p-\frac{1}{2})^2 \cdot k} = 2^{-\Omega(k)}$$

Let q be any polynomial. For $k \geq c \cdot q(n)$ (c is a suitable constant), one gets for both error probabilities $\leq 2^{-q(n)}$. □

Rule-non-determinsm. Probabilistic ASMs can also be defined for rule-non-determinism. The definition is done analogously to the one for the element-non-determinism. The (random) sequence X consists of Boolean values. The components are independent again, and for every every component, the probability is 0.5 that true is the value in that component. So, with the same arguments, exactly the same propositions as for Turing machines hold.
Probabilistic ASMs for rule-non-determinism can also be considered as special ASMs for element-non-determinism. From this, one gets that the propositions from element-non-determinism hold (restricted to this special case). Even more propositions hold because the problem of non-empty sets with probability zero is eliminated.

Special case. In the special case of a vocabulary where the static part is equal to obligatory part, one can restrict attention to a special probability measure. Even if there are more, this is the most reasonable one. The following measure can be defined because the living part of the structure is known to ne a priori. In the general case, there is no possibility to define a probability measure in this way because, one does even nor know whether the living part is finite, countable or even uncountable. So, in the special case, take the probability measure μ for I defined by the following:
Let imported(I, P) be the set of all imported elements (where always only one possiblity is considered because the reserve can be identified with the natural numbers with the usual successor function). If imported(I, P) is finite then $\mu(\{a\}) = \frac{1}{|\text{Active}(I)|+|\text{imported}(I,P)|}$ for all $a \in \text{Active}(I) \cup \text{imported}(I, P)$

Chapter 8

Simulating Other Computation Models by ASMs

Until now, complexity theory for ASMs has been considered. Many of the results are similar to those for other computation models. So, the question for the connection between ASMs and other computation models arises. In this chapter, three other computation models will be considered and it will be shown, how they can be simulated by ASMs.
First, define the notion of simulation which will be used in the following.

Definition 8.1. A machine $\mathcal{B}$ simulates a machine $\mathcal{A}$ in lock-step with lag factor c if there exists a mapping Φ from the states of $\mathcal{A}$ to the states of $\mathcal{B}$ such that for every run $(A_i)_{i\in\Lambda}$ of $\mathcal{A}$, there is a run $(B_i)_{i\in\mathbb{N}}$ of $\mathcal{B}$ and a monotone function $J : \Lambda \to \mathbb{N}$ (it maps indices to indices) such that

1. $J(0) = 0$
2. $B_{J(i)} = \Phi(A_i)$.
3. If an output β is emitted during the transition from A_i to A_{i+1}, then there is a unique $l \in [J(i), J(i+1))$ such that an output is emitted during the transition from B_l to B_{l+1}. Furthermore, this output is β. And if no output is emitted during any transition from A_i to A_{i+1} then no output is emitted during any transition from B_l to B_{l+1}, $l \in [J(i), J(i+1))$.
4. If $0 < i < \max\Lambda$ then $J(i) - J(i-1) \leq c$.
5. If Λ is finite, $i = \max(\Lambda)$, and A_i is final , then $B_{J(i)}$ is final.

$\mathcal{A}$ strictly lock step simulates $\mathcal{B}$ if $c = 1$.

This definition is the one from [6]. Because ASMs only emit an output in the last state and this is also true for the other computation models considered in this chapter, 3. can be reduced to: If A_i is a final configuration and β is the

corresponding output then β is also the output yielded by $B_{J(i)}$.
The states of Turing machines, RAMs and BSS-machines will be called configurations in the following to avoid a confusion with ASMs.
The output of an ASM is the last structure of a run if a Boolean output is recommended. Else, the value of Output is the ytelded output.

8.1 Turing Machines

First, the basic definition of Turing machines is recalled to use the denotation in the following.

Definition 8.2. A Turing machine (TM) is a 7-tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

- Q is a finite set of control states.
- Γ is a finite set of allowed band symbols.
- B is the blank symbol and it is an element of Γ.
- Σ is a set of input symbols, it is a subset of Γ which does not contain B.
- δ is the transition function, it is a mapping from $Q \times \Gamma$ to $Q \times \Gamma \times \{-1, 0, 1\}$ (δ might be undefined for some arguments).
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of the final control states.

An input for M is an element $w \in \Sigma^*$. The size of the input is the length of w and the time needed for the computation is the number of transitions done during the computation of M on w. Because it is assumed that this model is well known, there will not be any further explication about this definition or its meaning.
In [10], it has already been roughly described, how TMs can be simulated by ASMs. This will be described in detail now.
A configuration can be described by the current control state, the current position of the head and the content of the tape.
The states of the simulating ASM are structures of the vocabulary $\{\text{currentControl}, \text{Head}, \text{content}, \text{final}, \text{Succ}, \text{Pred}\} \cup Q \cup \Sigma$ (furthermore it contains the Boolean function names and the nullary function Output, Mode). Succ and Pred are static functions. The elements of Σ and of Q are static constants. The background part of the structure is $\mathbb{N} \cup \Sigma \cup Q$ (where $\mathbb{N}, \Sigma^*, Q$ are are disjoint sets). The active part contains a finite subset of $\mathbb{N}$, Q and Σ. The exact meanings of the functions are the following:

Succ: This the usual successor function for $\mathbb{N}$.

Pred: This the usual predecessor function for $\mathbb{N}$.

currentControl: This nullary function gives the current control state of the simulated Turing machine (so, its value is an element of Q).

Head: This nullary function gives the current position of the head of the simulated Turing machine (so, its value is an element of $\mathbb{N}$).

content: This unary function gives for field of the tape of the simluated Turing machine its content. So, content : $\mathbb{N} \to \Sigma$. If the content of the i-th field is $\sigma \in \Sigma$ then content$(i) = \sigma$. If the content of the i-th field is the blank symbol then content$(i) =$ undef.

Mode is initialized by true. Output is only important in the case of a Boolean output.
The active part of the universe respectively consists of those natural numbers i such that the i-th position in the tape does not contain the blank symbol, the current contol state, the current head position, the current value of Output and those elements written on any position of the tape.
The set of all input structures for the simulating ASM is the set of all translations of the inputs of the translated Turing machine.
For every tuple $(q_1, \sigma_1, q_2, \sigma_2)$ such that $\delta(q_1, \sigma_1) = (q_2, \sigma_2, x)$ and $q_2 \notin F$ define the following rule:

$$
\begin{array}{l}
\texttt{if } \text{currentControl} = q_1 \texttt{ and } \text{content(Head)} = \sigma_1 \texttt{ then} \\
\quad \texttt{block} \\
\quad\quad \text{currentControl} := q_2 \\
\quad\quad \text{content(Head)} := \sigma_2 \\
\quad\quad \text{Head} := \begin{cases} \text{Pred(Head)} & : \quad x = -1 \\ \text{Head} & : \quad x = 0 \\ \text{Succ(Head)} & : \quad x = 1 \end{cases} \\
\quad \texttt{endblock} \\
\texttt{endif}
\end{array}
$$

For every tuple $(q_1, \sigma_1, q_2, \sigma_2)$ such that $\delta(q_1, \sigma_1) = (q_2, \sigma_2, x)$ and $q_2 \in F$ define the following rule:

```
if currentControl = q_1 and content(Head) = σ_1 then
  block
    currentControl := q_2
    content(Head) := σ_2
```

$$\text{Head} := \begin{cases} \text{Pred(Head)} & : \quad x = -1 \\ \text{Head} & : \quad x = 0 \\ \text{Succ(Head)} & : \quad x = 1 \end{cases}$$

```
    Mode := false
  endblock
endif
```

Let R_M be the sequence of all those rules described above. The ASM-program simulating the Turing machine looks as follows:

```
block
  R_M
endblock
```

Now, consider the complexity of this simulation. If I is the input structure (for the ASM) corresponding to the input for the Turing machine w then size(I) is equal to the length of w.
One transition done by the Turing machine corresponds one macrostep done by the ASM.
Non-determinism for Turing machines can be simulated by the use of the guess-rule rule.

So, in this section, the following theorem has been proven:

Theorem 8.3. *For every Turing machine M there exists an ASM that strictly lock-step simulates M.*

8.2 Random Access Machines

8.2.1 The Basic Model

First, the random access machine (RAM) model used in this work will be described. This is the model used in [1].
A RAM has a potentially infinite supply of registers, indexed by the non-negative integers, each capable of holding (as its content) an arbitrary non-negative integer. Register 0 plays a special role and is called the accumulator. The input to a RAM is a finite sequence of non-negative integers, initially stored in registers 1 through l for some l; the accumulator and all registers after l initially hold 0. The result of a computation is the content of the accumulator when (and if) a HALT operation is executed. (If a Boolean output is desired then use the parity

of the number in the accumulator.)
A RAM program is a list of instructions, each having the form "operation operand" where the operand is a non-negative integer and the operation is one of the following, where it is also indicated how to execute each instruction when the operand is a. The phrase "Put t into register i" means to change the content of register to be t while leaving the contents of all other registers unchanged; $c(i)$ means the content of register i.

LOAD= Put a into the accumulator.

LOAD Put $c(a)$ into the accumulator.

LOAD* Put $c(c(a))$ into the accumulator.

STORE Put $c(0)$ into register a.

STORE* Put $c(0)$ into register $c(a)$.

ADD= Add a to the accumulator, i.e., put $c(0) + a$ into the accumulator.

ADD Add $c(a)$ to the accumulator.

ADD* Add $c(c(a))$ to the accumulator.

SUB= Subtract a from the accumulator and if the result is negative then replace it with 0.

SUB Subtract $c(a)$ from the accumulator and if the result is negative then replace it with 0.

SUB* Subtract $c(c(a))$ from the accumulator and if the result is negative then replace it with 0.

MULT= Multiply the accumulator by a.

MULT Multiply the accumulator by $c(a)$.

MULT* Multiply the accumulator by $c(c(a))$.

DIV= Divide the accumulator by a and round down to the next smaller integer.

DIV Divide the accumulator by $c(a)$ and round down to the next smaller integer.

DIV* Divide the accumulator by $c(c(a))$ and round down to the next smaller integer.

JUMP Continue with the instruction in position a in the program.

JGTZ If $c(0) > 0$ then continue with the instruction in position a; otherwise continue with the next instruction after the current one.

JZERO If $c(0) = 0$ then continue with the instruction in position a; otherwise continue with the next instruction after the current one.

HALT Halt.

It is assumed that the last instruction in a program is always HALT, with an (irrelevant) operand of 0.
A configuration of a RAM consists of the content of its registers, its program, and a program counter pointing to the next instruction in the program to be executed. It is an initial state if register 0 and all but finitely many of the other registers contain 0 and the program counter points to the first instruction in the program. It is a final state if the program counter points to a HALT instruction. It is a hang state if the next instruction would require division by zero or continuing with a non-existing instruction. In all enabling state, after the next instruction is executed, the program counter is moved forward one step in the program except as specified in the instructions JUMP, JGTZ, and JZERO.
Now, the usual time measure will be described. The length of an input is the last register with non-zero content (or 1 if there is no such register), and the time needed by a computation is the number of instruction executions in it. This is called the "unit cost measure".

8.2.2 Simulating RAMs

Vocabulary, Background Structure, Input Structure

In this section, the vocabulary, the background structure and the input structure of the simulating ASM are described (and the intended interpretation).
The active part of the universe consists of a finite subset of $\mathbb{N}$. This is the set of all non-negative integers i such that: $i \neq 0$ is the content of a register or i is the current instruction.
First, the vocabulary contains those elements of $\mathbb{N}$ as constants which are explicitly named in the program of the RAM.
Furthermore, the vocabulary Υ contains the following dynamic function names:

currentInst: For every state, this nullary function (name) gives the number of the current instruction in the program of the RAM. Initially, its value is zero.

content: This is a unary function (name) giving for each register (identified with a natural number) its content in the current state. For $i \in \mathbb{N}$, if $x_i = n$ for an $n \in \mathbb{N}, n \neq 0$ then content$(i) = n$; if $n = 0$ then content$(i) =$ undef. (It is necessary to use undef here because, else, the resulting structures would not be of finite size.

Now, consider the background structure:

1. The universe (called background part in the preceding chapters) is $\mathbb{N}$.

2. The static part of the vocabulary (except for the logical part) consists of:

   (a) The constant $0 \in \mathbb{N}$ is a static (nullary) function.

   (b) The addition $+$ (2-ary), which is defined as usual on $\mathbb{N}$, is a static function.

   (c) A kind of subtraction SUB (2-ary) is a static function. For $x, y \in \mathbb{N}$, x SUB y is defined as follows: If $x - y$ (defined as usual for the natural numbers is non-negative) then x SUB $y = x - y$ else x SUB $y = 0$

   (d) multiplication $\cdot$ (2-ary), which is defined as usual on $\mathbb{N}$, is a static function.

   (e) A kind of division DIV (2-ary) is a static function. For $x, y \in \mathbb{N}$, x DIV y is defined as x DIV $y = \lfloor \frac{x}{y} \rfloor$.

   (f) The order $<$ on $\mathbb{N}$ (to be more precise, this is a 2-ary function tagged relational), defined as usual, is an element of the static part of the vocabulary.

   (g) The successor function Succ according to $<$ is a static function.

   (h) If a Boolean output is desired then a function giving the parity of each integer has to be added to the set of the static functions.

State Mapping

As demanded in definition 8.1, a mapping Φ will be defined which maps configurations of a RAM to states of an ASM.
For a configuration $A := (k, x, Q)$ (k gives the current instruction index, x gives the content of the registers, Q is the program) of the RAM, the corresponding state of the ASM $\Phi(A)$ is defined as follows (the program Q will be coded in the ASM-program):

1. The vocabulary and the background structure of the ASM state are as already described. Therefore, they do not depend on the RAM state which is mapped.

2. If register i contains the natural number $n > 0$ then content$(i) = n$. If register i contains 0 then content$(i) =$ undef.

3. currentInst $= k$

4. If the RAM executes the HALT-instruction, then Mode is equal to false. In the states before, Mode $=$ true.

5. If register 0 contains $n \in \mathbb{N}$ and the current instruction is a HALT instruction then Output $= n$.

The set of all input structures for the simulating ASM is the set of all translations of the inputs of the translated RAM.

Simulating RAM programs

In this section, it will be described, how a RAM program can be translated into an ASM-program.
First, the input state for the ASM is described. Mode must be set to true. currentInst is set to 0. content is set as given by the input.
Each instruction of the RAM will be translated as a conditional rule. The condition tests the index of the instruction for respectively executing the right instruction.
Now, the different types of instructions will be considered and a translation will be given for all of them. Let halt be the set of all instruction indexes k such that the k-th instruction is HALT.
The translation is always a direct consequence of the definition of the respective instruction. There is only one thing, one has to care for. If the content of register is 0, then content(i) = undef in the respective state of the ASM. So, different cases have to be considered and partially undef has to be substituted by 0. Only the cases, where the update changes a location, have to be named in the program. Not that a is the operand.

k **: LOAD=** is translated into:

```
if currentInst = k then
  block
    content(0) := a
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k **: LOAD** is translated into:

```
if currentInst = k then
  block
    if content(a) ≠ undef then content(0) := content(a)
    else content(0) := 0
    endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k **: LOAD*** is translated into:

```
if currentInst = k then
  block
    if content(a) ≠ undef ∧ content(content(a)) = undef
    then content(0) := 0 endif
    if content(a) ≠ undef ∧ content(content(a)) ≠ undef
    then content(0) := content(content(a)) endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k **: STORE** is translated into:

```
if currentInst = k then
  block
    content(a) := content(0)
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k **: STORE*** is translated into:

```
if currentInst = k then
  block
    if content(a) ≠ undef
    then content(content(a)) := content(0) endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k : **ADD=** is translated into:

```
if currentInst = k then
  block
    if content(0) ≠ undef then content(0) := content(0) + a
    else content(0) := a endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k : **ADD** is translated into:

```
if currentInst = k then
  block
    if content(0) = undef
    then content(0) := content(a) endif
    if content(0) ≠ undef ∧ content(a) ≠ undef
    then content(0) := content(0) + content(a) endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k : **ADD*** is translated into:

```
if currentInst = k then
  block
    if content(a) = undef ∧ content(0) ≠ undef
    then content(0) := content(0) + content(0) endif
    if content(a) ≠ undef ∧ content(content(a)) ≠ undef
        ∧content(0) = undef
    then content(0) := content(content(a)) endif
    if content(a) ≠ undef ∧ content(content(a)) ≠ undef
        ∧content(0) ≠ undef
    then content(0) := content(0) + content(content(a)) endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k **: SUB=** is translated into:

```
if currentInst = k then
  block
    if content(0) ≠ undef
    then content(0) := content(0) SUB a endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k **: SUB** is translated into:

```
if currentInst = k then
  block
    if content(0) ≠ undef ∧ content(a) ≠ undef then
    content(0) := content(0) SUB content(a) endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k **: SUB*** is translated into:

```
if currentInst = k then
  block
    if content(a) = undef ∧ content(0) ≠ undef
    then content(0) := undef endif
    if content(a) ≠ undef ∧ content(content(a)) ≠ undef
        ∧content(0) ≠ undef
    then content(0) := content(0) SUB content(content(a))endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k : **MULT=** is translated into:

```
if currentInst = k then
  block
    if content(0) ≠ undef
    then content(0) := content(0) · a endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k : **MULT** is translated into:

```
if currentInst = k then
  block
    if content(0) ≠ undef ∧ content(a) ≠ undef
    then content(0) := content(0) · content(a) endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k : **MULT*** is translated into:

```
if currentInst = k then
  block
    if content(a) = undef ∧ content(0) ≠ undef
    then content(0) := content(0) · content(0) endif
    if content(a) ≠ undef ∧ content(content(a)) ≠ undef
         ∧content(0) ≠ undef
    then content(0) := content(0) · content(content(a)) endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k **: DIV=** is translated into (note that division by zero indicates a hanging state):

```
if currentInst = k then
  block
    if content(0) ≠ undef
    then content(0) := content(0) DIV a endif
    if content(0) = undef
    then content(0) := content(0)DIV0 endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k **: DIV** is translated into (note that division by zero indicates a hanging state):

```
if currentInst = k then
  block
    if content(0) = undef
    then content(0) := content(a) DIV 0 endif
    if content(0) ≠ undef ∧ content(a) ≠ undef
    then content(0) := content(0)DIVcontent(a) endif
    currentInst := Succ(currentInst)
    if ⋁_{l∈halt} Succ(currentInst) = l
    then Mode := false endif
  endblock
endif
```

k **: DIV*** is translated into:

`if` currentInst $= k$ `then`
`block`
`if` $\text{content}(a) = \text{undef} \wedge \text{content}(0) \neq \text{undef}$
`then` $\text{content}(0) := \text{content}(0)\ \text{DIV}\ \text{content}(0)$ `endif`
`if` $\text{content}(a) \neq \text{undef} \wedge \text{content}(\text{content}(a)) \neq \text{undef}$
$\wedge \text{content}(0) \neq \text{undef}$
`then` $\text{content}(0) := \text{content}(0)\ \text{DIV}\ \text{content}(\text{content}(a))$ `endif`
$\text{currentInst} := \text{Succ}(\text{currentInst})$
`if` $\bigvee_{l \in \texttt{halt}} \text{Succ}(\text{currentInst}) = l$
`then` $\text{Mode} := \text{false}$ `endif`
`endblock`
`endif`

k **: JUMP** is translated into:

`if` currentInst $= k$ `then`
`block`
$\text{currentInst} := a$
`if` $\bigvee_{k \in halt} a = k$ `then` $\text{Mode} := \text{false}$ `endif`
`endblock`
`endif`

k **: JGTZ** is translated into:

`if` currentInst $= k$ `then`
`block`
`if` $\text{content}(0) > 0$ `then`
`block`
$\text{currentInst} := a$
`if` $\bigvee_{k \in halt} a = k$
`then` $\text{Mode} := \text{false}$ `endif`
`endblock`
`else`
`block`
$\text{currentInst} := \text{Succ}(\text{currentInst})$
`if` $\bigvee_{l \in \texttt{halt}} \text{Succ}(\text{currentInst}) = l$
`then` $\text{Mode} := \text{false}$ `endif`
`endblock`
`endif`
`endblock`
`endif`

k **: JZERO** is translated into:

```
if currentInst = k then
  block
    if content(0) = undef then
      block
        currentInst := a
        if ⋁_{k∈halt} a = k
         then Mode := false endif
      endblock
    else
      block
        currentInst := Succ(currentInst)
        if ⋁_{l∈halt} Succ(currentInst) = l
        then Mode := false endif
      endblock
    endif
  endblock
endif
```

k **: HALT** Here, nothing has to be done. This has already been treated by defining the set halt.

This ASM obviously simulates a given RAM in the sense of definition 8.1. This can be proven analogously to theorem 8.9.
The size of the input structure for the simulating ASM is less or equal to the size of the input for RAM. These two are equal if after the first register containing zero, all registers contain zero. One time unit (measured with the described time measure as described above) is simulated by one macrostep.

Definition 8.4. A *non-deterministic RAM* is a RAM that can non-deterministically guess tuples over $\mathbb{N}$.

Non-determinism can be simulated by use of the choose-element rule.

So, in this section, the following theorem has been proven:

Theorem 8.5. *For every RAM M there exists an ASM that strictly lock-step simulates M.*

8.3 BSS-Machines

Another computation model well known from literature are BSS-machines over $\mathcal{R}$ where $\mathcal{R}$ is a commutative ring with unit and it is supposed that $\mathcal{R}$ is ordered. Roughly spoken, a BSS-machine over $\mathcal{R}$ is a RAM over $\mathcal{R}$. It works with the structure $(\mathcal{R}, +, -, \cdot, <, (c_r)_{r\in\mathcal{R}})$ (if $\mathcal{R}$ is a field then the vocabulary does also contain /). Note that the vocabulary might be infinite but later, a program will only use finitely many constants. So, one can see a certain similarity of the ideas behind this model and the idea of ASMs. Especially, the question arises whether ASMs can simulate BSS-machines. As one will see later, this is possible. So, the ASMs are at least as powerful as BSS-machines.

8.3.1 The Basic Model

First, the BSS-model is described. This definition is equivalent to the one given in [5] up to an increase of the running time by a constant factor. There it is defined via graphs, computation nodes ... .

Definition 8.6. $\mathcal{R}^\infty$ is defined as the set of all infinite $\mathcal{R}$-sequences such that all but finitely many elements are equal to zero.
Let $w = w_0 w_1 w_2 ... \in \mathcal{R}^\infty$. Then the length of w length(w) is defined as the minimal number k such that $w_n = 0$ for all $n \geq k$.

The following two formulations as are equivalent to the preceding definition for $\mathcal{R}^\infty$:

- $\mathcal{R}^\infty := \bigcup_{i=1}^{\infty} \mathcal{R}^i$ where $\mathcal{R}^i$ is the set of all i-tuples over $\mathcal{R}$.
- $\mathcal{R}^\infty$ is the set of all function $f : \mathbb{N} \to \mathcal{R}$ such that $f(n) = 0$ for all but finitely many $n \in \mathbb{N}$.

Definition 8.7. Let $n, m \in \mathbb{N}$. A map $f : \mathcal{R}^n \to \mathcal{R}^m$ is a *polynomial map* if the coordinate maps f_i are polynomial in the n variables for $i \in \{1, ..., m\}$. If $\mathcal{R}$ is a field, then f is rational if the f_i are rational functions in the n variables.

Based on this, the deterministic version of the BSS-machine over $\mathcal{R}$ is defined.

Definition 8.8. A BSS-machine M over $\mathcal{R}$ is given by a finite set I of instructions labeled by $0, ..., N$. The input and output spaces are subsets of $\mathcal{R}^\infty$. A configuration is a quadrupel $(k, r, w, x) \in I \times \mathbb{N} \times \mathbb{N} \times \mathcal{R}^\infty$, where k is the instruction currently executed, r and w are the numbers of the so called "copy registers" and x describes the content of the registers of the machine. On input $x \in \mathcal{R}^\infty$, the computation is started with configuration $(0, 0, 0, x)$. If $k = N$ the computation stops; in that case the value of x is the output computed by the machine. The instructions are of the following types:

computation: $k : x_0 \leftarrow g_k(x)$ where g_k is a polynomial function on $\mathcal{R}^\infty$ or a rational function if $\mathcal{R}$ is a field. The value indicated by $g_k(x)$ is allocated to x_0. The next instruction will be $k+1$, the copy registers are updated according to either $r \leftarrow r+1$ or $r \leftarrow 0$ and similarly for w. The other registers remain unchanged.

branch: k : **if** $x_0 \geq 0$ **goto** $\beta(k)$ **else goto** $k+1$. In the case of $x_0 \geq 0$, the next instruction will be $\beta(k)$. In the case of $x_0 < 0$, the next instruction will be $k+1$. The contents of the registers remain unchanged.

copy: k : $x_w \leftarrow x_r$, i.e., the content of the "read-register" is copied into the "write-register". The next instruction is $k+1$; all other registers remain unchanged.

g_k depends on a finite set of variables $\{x_{i_1}, ..., x_{i_s}\}$ such that $\{x_{i_1}, ..., x_{i_s}\} \subseteq \{0, ..., \text{length}(x) - 1\}$.
The time needed by a computation is defined as the number of executed instructions during the computation.
Up to an increase of running time by a constant factor, the power of BSS-machines does not change if one restricts the computation-instructions to the basic operations $+, -, \cdot$ (and / if $\mathcal{R}$ is a field) and assignments $x_0 \leftarrow \alpha$ where α belongs to a fixed finite set of elements of $\mathcal{R}$.
For BSS-machines, the notion of a run and the notion of a final configuration are defined as for ASMs.

8.3.2 Simulating BSS-machines

It will be shown that BSS-machines can be lockstep-simulated by ASMs according to definition 8.1. So, in this section, the following theorem will be shown:

Theorem 8.9. *For every BSS-machine $\mathcal{A}$ over $\mathcal{R}$ there exists an ASM that strictly lock-step simulates $\mathcal{A}$.*

Vocabulary, Background Structure, Input Structure

In this section, the vocabulary, the background structure and the input structure of the simulating ASM are described (and the intended interpretation).
The active part of the universe consists of a finite set $\{0, ...n\} \subseteq \mathbb{N}$. n is the maximum of (the maximal instruction index) and (the maximal number of registers -1) used by the BSS-machine. More precisely, the active part contains the current instruction index and the numbers of those registers which contain an element of $\mathcal{R}$ not equal to 0. Furthermore, it contains those elements of $\mathcal{R}$ unequal to zero which are currently written in one of the registers.
First, the vocabulary contains those elements of $\mathcal{R}$ as constants which are explicitly named in the program of the BSS-machine, i. e., there exists a rational function in the program which uses this element as a coefficient. Furthermore, the vocabulary Υ contains the following dynamic function names:

currentInst: For every state, this nullary function (name) gives the number of the current instruction in the BSS-machine program. Initially, its value is zero.

content: This is a unary function (name) giving for each register (identified with a natural number) its content in the current state. In detail, this means the following:
For $i \in \mathbb{N}$, if $x_i \neq 0$ and $x_i = r$ then content$(i) = r$ and if $x_i = 0$ then content$(i) =$ undef.

read: This is a constant (from $\mathbb{N}$). It is set correspondingly to r in the configuration of the BSS-machine and initially set to 0.

write: This is a constant (from $\mathbb{N}$). It is set correspondingly to w in the configuration of the BSS-machine and initially set to 0.

Now, consider the background structure:

1. The universe (called background part in the preceding chapters) is $\mathcal{R} \cup \mathbb{N}$ ($\mathcal{R} \cap \mathbb{N} = \emptyset$).

2. The static part of the vocabulary (except for the logical part) consists of:

   (a) For elements $x, y \in \mathcal{R}$, addition $x + y$ and multiplication $x \cdot y$ are defined as for the elements of $\mathcal{R}$. The order relation $<$ is defined in the same way.

   (b) The function Succ is the usual successor function on $\mathbb{N}$ (and is only defined on $\mathbb{N}$).

State Mapping

As demanded in definition 8.1, a mapping Φ will be defined which maps states of a BSS-machine to states of an ASM.
For a configuration $A := (k, w, r, x)$ of the BSS-machine, the corresponding state of the ASM $\Phi(A)$ is defined as follows:

1. The vocabulary and the background structure of the ASM state are as described before. Therefore, they do not depend on the BSS-machine state which is mapped.

2. If register i contains $r \in \mathcal{R} - \{0\}$ (i.e., $x_i = r, r \neq 0$) then content$(i) = r$. If register i contains 0 then content$(i) =$ undef.

3. currentInst $= k$

4. If the BSS-machine executes the STOP-instruction resp. the last instruction of the BSS-machine program, then Mode is set to false. In the states before, Mode $=$ true.

5. read $= r$, write $= w$

6. Output $= x$

The set of all input structures for the simulating ASM is the set of all translations of the inputs of the translated BSS-machine.

Simulating BSS-machine programs

In this section, it will be described, how a BSS-machine program can be translated into an ASM-program.
First, the input state for the ASM is described. content is set according to the input of the BSS-machine. Mode is set to true. Output is equal to the input of the BSS-machine. read and write are set to zero. currentInst is set to zero.
Each instruction of the BSS-machine will be translated as a conditional rule. The condition tests the index of the instruction for respectively executing the right instruction.
Now, the different types of instructions will be considered and a translation will be given for all of them.

$k : x_0 \leftarrow g_k(x)$ Let $\tilde{g}_k$ result from g_k as follows: If g_k depends $x_{i_1}, ..., x_{i_n}$ then $\tilde{g}_k$ computes the same value where the computation is based on the contents of corresponding registers.For simplicity, $\tilde{g}_k$ is said to depend on the register indexes. content(0), read, write are updated corresponding to the instruction of the BSS-machine.

```
if currentInst = k then
  block
    currentInst := Succ(currentInst)
    content(0) := g̃_k(i_1, ..., i_k)
    update for read
    update for write
    if Succ(currentInst) = N then Mode := false endif
  endblock
endif
```

where `update for` read is the following:

- for $r \leftarrow r + 1$:

  read := Succ(read)

- for $r \leftarrow 0$

  read := 0

`update for` write is defined analogously to `update for` read.

k **: if** $x_0 \geq 0$ **goto** $\beta(k)$ **else goto** $k+1$ This is translated by a conditional rule testing the condition $x_0 \geq 0$ (or to be more precise the translation of this condition). currentInst is set according to whether the condition is fulfilled or not and according to the instruction index indicated by `goto`.

```
if currentInst = k then
  block
    if content(0) = undef
        ∨(content(0) ≠ undef ∧ content(0) ≥ 0) then
      block
        currentInst := β(k)
        if β(k) = N then Mode := false endif
      endblock
    endif
    if ¬(content(0) = undef
        ∨(content(0) ≠ undef ∧ content(0) ≥ 0)) then
      block
        currentInst := Succ(currentInst)
        if Succ(currentInst) = N then Mode := false endif
      endblock
    endif
  endblock
endif
```

$k : x_w \leftarrow x_r$ The write-th component is updated by the content of the read-th component.

```
if currentInst = k then
  block
    content(write) := content(read)
    currentInst := Succ(currentInst)
    if Succ(currentInst) = N then Mode := false endif
  endblock
endif
```

instruction counter equal to N**:** In the step before, Mode has already been set to false. So, nothing has to be done in this case.

If a Boolean output is expected then the nullary function Output has to be updated accordingly in the last step.

Proof of Theorem 8.9

To prove theorem 8.9, the following lemma will be proven first. Theorem 8.9 is a direct consequence of lemma 8.10.

Lemma 8.10. *Let $A_0, A_1, ...$ be a run of a BSS-machine $\mathcal{A}$. Let $\mathcal{B}$ be the ASM-program which results from translating $\mathcal{A}$ to an ASM-program as described above. Let $B_0, B_1, ...$ be a run of $\mathcal{B}$ such that $B_0 = \Phi(A_0)$. Then for every i such that i is less or equal to the length of the run, $B_i = \Phi(A_i)$.*

Proof. The proof is done by induction on the index of the states of the BSS-machine run.
For $i = 0$, $B_i = \Phi(A_i)$ is given by the assumption made in the lemma.
Let $B_n = \Phi(A_n)$ for an $n \in \mathbb{N}$.
Now, consider $n + 1$. A_{n+1} is obtained from A_n by executing the translation of the instruction determined by the value of currentInst in A_k. For the type of this instruction there are four possiblities. Each case (for a BSS-machine instruction) will be considered in the following.

$k : x_0 \leftarrow g_k(x)$**:** x_0 is updated in A_n according to g_k. The same is done with content(0). read, write are updated in the same way as r, w. The instruction counter of the BSS-machine is increased. currentInst is also increased. Output is updated according to the update of the registers.

k **: if** $x_0 \geq 0$ **goto** $\beta(k)$ **else goto** $k + 1$**:** This causes, that in the case of $x_0 \geq 0$ the instruction counter of the BSS-machine is set to $\beta(k)$. In B_i currentInst is also set to $\beta(k)$. In the case of $\neg(x_0 \geq 0)$, the instruction counter of the BSS-machine is increased by 1. The same is done with currentInst.

$k : x_w \leftarrow x_r$**:** In A_i, the w-th component gets the value of the r-th component. In B_i the same update is executed for content(write). Furthermore, the instruction counter of the BSS-machine and currentInst are increased both. Output is updated according to the update of the registers.

instruction counter equal to N**:** In the ASM, Mode is already set to false and Output does contain the right value. In the preceding instruction, it has already been tested whether the instruction counter is set to N. If so, then Mode is set to false.

Because an instruction resp. its translation is executed iff currentInst has the value of this instruction, by the induction hypothesis one gets that $B_{n+1} = \Phi(A_{n+1})$. □

Definition 8.11. A *non-deterministic BSS-machine* is a BSS-machine that can non-deterministically guess tuples $Y \in \mathcal{R}^\infty$.

The non-determinism (introduced via the possibility of guessing elements from $\mathcal{R}^\infty$) can be simulated by using the guess-element rule `guess-element` $v \; : \; v \in \mathcal{R}$ (the guard $v \in \mathcal{R}^\infty$ has to be replaced by back(v) which is equivalent by the definiton of the unary relation back which marks the elements of the background part).

8.3.3 Remark

In [15], an extension of Turing machines is introduced. This extended model does not only work (or another alphabet) and the operations are not only writing 0 or 1 but there is any arbitrary structure behind (description of this idea, page 100). The same can be done with BSS-machines. The structure behind a BSS-machine is always an ordered ring and the operations are limited to addition, subtraction, multiplication (in the case of a field, division is also an operation) and writing into a register. The tested conditions are limited to testing $= 0, < 0, > 0, \leq 0, \geq 0$.
This can be generalized in a similar way as for Turing machines. Put an arbitrary structure behind with vocabulary Υ ($\Upsilon = \Upsilon_{\text{relational}} \cup \Upsilon_{\text{functional}}, \Upsilon_{\text{relational}} \cap \Upsilon_{\text{functional}} = \emptyset$). Instead of addition, subtraction and multiplication (and division) any arbitrary function can be used. So, $k : x_0 \leftarrow g_k(x)$ (g_k polynomial (rational)) is replaced by $k : x_0 \leftarrow g_k(x)$ where $g_k(x)$ is any term constructed from variables and elements of $\Upsilon_{\text{functional}}$. k : `if` c `then goto` $\beta(k)$ where $c \in \{x_0 = 0, x_0 < 0, x_0 > 0, x_0 \leq 0, x_0 \geq 0\}$ is replaced by k : `if` $\phi(x)$ `then goto` $\beta(k)$ where $\phi(x)$ is an FO-formula without quantors over Υ. The rest of the BSS-model does not change.
The simulation of this computation model by ASMs can be done as for "ordinary" BSS-machines. The background structure for the states of the simulating ASM is the the structure behind the simulated BSS-machine. The lock-step simulation can be done as in the described case.
Note that the properties of the computation model strongly depends on the structure behind, e.g. the relation between non-deterministic polynomial time and deterministic polynomial time.

8.4 Consequences

So, Turing machines, RAMs and BSS-machines can be considered as special ASMs. Furthermore, the complexity does not change. To be more precise, this will be explained now.
The complexity measures for the computation models named above (Turing machine, RAM, BSS-machine) have already been defined. The size of the input structure of the simulating ASM is equal to the size of the input of the Turing

machine resp. $\leq$ the RAM resp. $\leq$ the BSS-machine. The time needed for the computation of the Turing machine resp. the RAM resp. the BSS-machine (using the respective time measure) is equal to the number of macrosteps done by the simulating ASM.

The first idea could be now that the theorems which hold for ASMs (where time is measured via macrosteps) could be directly transfered to Turing machines, RAMs and BSS-machines. But this is not true in that simple way. One example is for this the Speed-Up-Theorem (in its respective formulation). It holds for ASMs (where time is measured in macrosteps). Now, one could think that -as a consequence- it also holds for Turing machines and RAMs but this is not correct. It holds for Turing machines but not for RAMs (this is a direct consequence of the Linear Time Hierarchy Theorem for RAMs, proven in [1]). So, the reason, why the Speed-Up-Theorem holds for Turing machines, is another one and it is necessary to prove it separately. But indeed, there is a connection between the Speed-Up-Theorem for Turing machines and the one for ASMs. This is given by the *way* of proving them. In the prove of the Speed-Up-Theorem for Turing machines, for a Turing machine M, a Turing machine M' is constructed that simulates M within the time $c \cdot f(n)$. Therefore, each symbol of M' encodes several symbols of M. As a result, M' can simulate several moves of M by one move of its own. In the prove of the Speed-Up-Theorem for ASMs (where time is measured in macrosteps), for an ASM-program Q an ASM-program Q' is constructed that can simulate several macrosteps of Q by one of its own. For RAMs such a construction is not possible.

Now, consider further results.

Obviously, hardness-results for ASMs can be adopted directly. To be more precise, this means the following:

For every complexity class in one computation model, there is a corresponding one in the other computation models respectively. First, this notion will be defined more precisely. For determinism, this is rather simple. If C is the set of all languages recognizable by a TM resp. RAM resp. BSS-machine within the given (deterministic) time $\leq f(n)$ (using the respective time measure) then the corresponding complexity class (for ASMs) is $\tilde{C} = \mathrm{TIME}^{\mathrm{macro}}(f(n))$. For non-determinism, this is rather the same but the kind of non-determinism has to be considered. If C is the set of all languages recognizable by a TM within the given non-determinstic time $\leq f(n)$ (using the time measure for TMs) then the corresponding complexity class (for ASMs) is $\tilde{C} = \mathrm{NTIME}^{\mathrm{macro}}_{\mathrm{rule}}(f(n))$ because, for the simulation of a non-deterministic TM, only the choose-rule rule is necessary (as non-deterministic rule). If C is the set of all languages recognizable by a RAM resp. a BSS-machine within the given non-determinstic time $\leq f(n)$ (using the respective time measure) then the corresponding complexity class (for ASMs) is $\tilde{C} = \mathrm{NTIME}^{\mathrm{macro}}_{\mathrm{el}}(f(n))$.

This is defined analogously for limitations concerning space.

For ASMs, a lot of theorems do not hold for a complete complexity class but for a

restriction. These restrictions are e.g. an ordered active part or additional functions which are defined nowhere. These restrictions do not need to be considered for the complexity classes for TMs, RAMs, BSS-machines which correspond to the complexity classes of ASMs.
E.g. $\mathrm{P}^{\mathrm{macro}}$ ($= \mathrm{P}$) for ASMs corresponds to P for Turing-machines, $\mathrm{NP}^{\mathrm{macro}}_{\mathrm{rule}}$ ($= \mathrm{NP}_{\mathrm{rule}}$) for ASMs corresponds to NP for Turing-machines and so on. The same can be done for RAMs and BSS-machines. Now, consider a complexity class C for any of these three computation models (TMs, RAMs, BSS-machines) and let $L \in C$. So, there exists a TM resp. RAM resp. BSS-machine which accepts L within the limitations given by C. So, there exists a pair $(\tilde{L}, \mathrm{Input})$ of sets of Υ-structures (the first is the set of structures which comes from translating the elements of L according to this chapter, Υ is defined as described in the respective case; Input is the set of all translations of the inputs) and an ASM $\mathcal{A} = (Q, \mathrm{Input})$ such that $\tilde{L} = \mathrm{Acc}(Q)$ and $\mathcal{A}$ works within the limitations corresponding to $\tilde{C}$, so $\tilde{L} \in \tilde{C}$ ($\tilde{C}$ is the complexity class for ASMs corresponding to C).
If $(\tilde{L}, \mathrm{Input})$ is $\tilde{C}$-hard (via a given reduction) then L is C-hard (via the corresponding reduction). For example, the NP-hardness (where NP is the complexity class for Turing machines) of SAT (which is contained in Cook's Theorem) is a consequence of theorem 4.58. In [7] (which treats BSS-machines over $\mathbb{R}$) it has been shown that the algebraic-circuit problem is hard for the class $\mathrm{NP}_{\mathbb{R}}$ under $\mathrm{FO}_{\mathbb{R}}$-reductions (which has been defined in [7]). This is a consequence of theorem 4.55.
Completeness results can not be transfered that directly because the membership in the complexity classes can not be concluded directly. ASMs in general are more powerful than Turing machines resp. RAMs resp. BSS-machines and if $\tilde{L} \in \tilde{C}$ then it is not necessarily true that $L \in C$. The ASM (constructed as described) needs only one macrostep to simulate one step of the simulated machine but the other direction does not hold. So, membership has to be proven in every case.
For the time hierarchy and space hierarchy, the same things can be said as for the Speed-Up-Theorem. The methods are similar but the details are not. For example, the encodings are different.
The other kind of results are relations between complexity classes. I.e., given two complexity classes (for ASMs) $\tilde{C}_1$, $\tilde{C}_2$ then $\tilde{C}_1 \subseteq \tilde{C}_2$. This relation can not be concluded for C_1 and C_2. Else, one could also conclude the Speed-Up-Theorem for RAMs (which also contains a proposition about inclusions of complexity classes). The reason for not being able to make such conclusions is the same as why the membership in a complexity class can not be concluded. An important point to mention is again the similarity in the proofs for similar propositions.
The last point is descriptive complexity theory. In this work, a proposition, which is analogous to the one of Fagin's Theorem, has been proven (instead of second order logic, an extension of second order logic has been defined). The one direction of Fagin's theorem "$\mathrm{NP} \subseteq \Sigma^1_1$" (which is the more difficult direction) is a conse-

quence of theorem 4.49. Let $L \in$ NP. So, there exists a non-deterministic Turing machine M such that L is accepted by M within non-deterministic polynomial time. Let $\mathcal{A} = (Q, \text{Input})$ be the ASM simulating M where Input is the set of all structures corresponding to an input of M. Let $\tilde{L}$ be the set of input structures (over the vocabulary Υ) corresponding to L. Then $\tilde{L}$ is accepted by $\mathcal{A}$ within non-deterministic polynomial time (where only the rule-non-determinism is used). The rule-non-determinism can be simulated by element-non-determinism as already described. So, $\tilde{L}$ is accepted by an ASM $\mathcal{A}' = (Q', \text{Input})$ in polynomial non-deterministic time (where for the non-determinism only the guess-element rule is used). According to theorem 4.49, there exists a stat-Σ^1_1 sentence ψ such that $\tilde{L} = \{I | I \in \text{Input and } I \models \psi\}$. So, the same holds for L and in this special case, only a fragment of stat-Σ^1_1 is used which corresponds to Σ^1_1. So, the inclusion NP $\subseteq \Sigma^1_1$ is a consequence of theorem 4.49. The other inclusion of Fagin's Theorem can not be concluded in that way but it can be proven similar to the prove of stat-$\Sigma^1_1 \subseteq \text{NP}_{\text{el}}$.
Furthermore, in [7], a theorem analogous to Fagin's Theorem has been shown for BSS-machines over $\mathbb{R}$. The same as for Fagin's Theorem holds in this case and the reasons are also the same.
So, there does exist a connection between the complexity theory for ASMs and the complexity theory for the other computation models. But one has to be careful not to consider only the simplest way as one might be misleaded to because of the similarity of most of the results for the different computation models. In many cases, the simplest way is not really a way. Often, a direct transfer is not possible but an application of similar methods is possible and leads to the intentded results.

Chapter 9

Conclusions

The bsis for complexity theory via ASMs is to define the size of an input which is a structure. Based on this, complexity classes can be defined. The type a complexity class is given by the type of the accepting ASMs (deterministic, element-non-deterministic, rule-non-deterministic, alternating, probabilistic) and the complexity measure (time measures: macrosteps, microsteps; space).
The greatest problem occuring in this context is the following: Even if the size of structure is finite, the living part of the structure might be infinite. The reserve is always infinite but this is no problem because the reserve is a naked set.
Consequences

- One consequence comes up with element-non-determinism. Here a living element can be guessed (satisfying a Boolean- valued term). The set of all possibly guessed elements is not finite in general. Therefore, a state can have infinitely many non-isomorphic successors concerning a given program. So, in the general case, there is no limit for the number of successors of a structure.

- Infinitely many non-isomorphic states of the same size possible are in the general case. As a consequence, there are not many results on space complexity in the general case. The reason is that mostly, limits about the number of reachable within a given space are only possible in the special case that the static part of the considered vocabulary is restricted to the obligatory (i.e., logical) one.

- Alternating ASMs can be defined for both kinds of non-determinism. The problems are the same as already mentioned for non-determinism.

- For inputs of a probabilistc ASM, a probability measure has to be given One has to take into account that there might be non-empty sets with probability zero.

Because of the problems mentioned above, a special case of vocabularies (and therefore structures) is considered. Here, the background structure is reduced to the obligatory one (logic functions) and therefore, the size of the structre and the structure itself are finite. In this special case the problems mentioned above are eliminated.
The problem(s) mentioned above apart from those dealing with do not appear for determinism or rule-non-determinism because determinstic and rule-deterministic ASMs can only deal with those element which are accessable within a considered time. This gives another problem. Most propositions require that all active elements are accessible. This is not guaranteed in general. Therefore, a lot of propositions are restricted to structures whose active part is ordered by a successor function. This is a rather natural restriction.
To show inclusions of complexity classes, mostly some additional requirements on the vocabulary and the input structures are necessary. E.g., the active part has to ordered by a successor function.
At the end of this work, a comparison to other computation models, namely Turing machines, RAMs and BSS-machines. ASMs strictly lock-step simulate the other considered computation models. Complexity theory via ASMs is connected to complexity theory via one of the other computation models. But one has to be careful. Often, the results can not be transferred directly but the results are similar and the methods for proving are similar.
Thus, it turns out that it makes sense to introduce complexity theory via ASMs.

As a conclusion, one can say the following:
It is possible and does make sense to introduce complexity theory via Abstract State Machines. The ASM-model is a very powerful computation model but compexity theory does not too complicated.

Bibliography

[1] Andreas Blass and Yuri Gurevich, "The Linear Time Hierarchy Theorems for Abstract State Machines and RAMs", Journal of Universal Computer Science, vol. 3, no. 4 (1997), 247–278.

[2] Andreas Blass and Yuri Gurevich, "Fixed-Choice and Independent-Choice Logics", Journal of Symbolic Logic, to appear.

[3] Andreas Blass, Yuri Gurevich and Saharon Shelah, "Choiceless Polynomial Time", Microsoft Research Technical Report MSR-TR-99-08.

[4] Lenore Blum, Felipe Cucker, Michael Shub, Steve Smale, "Complexity and Real Computation", Springer Verlag, 1997; ISBN: 0-387-98281-7.

[5] Lenore Blum, Michael Shub, Steve Smale, "On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines", Bulletin of the American Mathematical Society, vol. 21, 1989, 1-46.

[6] Scott Dexter, Patrick Doyle, and Yuri Gurevich, "Gurevich Abstract State Machines and Schönhage Storage Modification Machines", Journal of Universal Computer Science, vol. 3, no. 4 (1997), 279-303.

[7] Erich Grädel, Klaus Meer, "Descriptive Complexity Theory over the Real Numbers", Lectures in Applied Mathematics, vol. 32, 1996, 381-403.

[8] Yuri Gurevich,"Evolving Algebras 1993: Lipari Guide", Specification and Validation Methods, ed. E. Börger, Oxford University Press, 1995, 9–36.

[9] Yuri Gurevich, "May 1997 Draft of the ASM Guide", University of Michigan EECS Department Technical Report CSE-TR-336-97.

[10] Yuri Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms", ACM Transactions on Computational Logic, to appear 2000.

[11] Yuri Gurevich and James K. Huggins, "Equivalence Is In The Eye Of The Beholder", Theoretical Computer Science(179)1-2(1997), 353–380.

[12] John E. Hopcroft, Jeffrey D. Ullman, "Introduction to automata theory, languages and computation", Addison-Wesley Publishing Company, Inc.,1997; ISBN: 0-201-02988-X.

[13] Neil Jones, "Computability and Complexity; From a Programming Perspective.", MIT PRESS, 1296 HC; ISBN: 0-262-10064-9.

[14] Christos Papadimitriou, "Computational Complexity", Addison-Wesley Publishing Company, Inc., 1994; ISBN: 0-201-53082-1.

[15] Bruno Poizat, "Les petits cailloux", Aléas Éditeur, juin 1995; ISBN: 2-908016-58-3.

Index

L_{inf}, 30, 72
Υ_{stat}, 5
Υ_{dyn}, 5
$\mathcal{R}^{\infty}$, 112
$\mathcal{T}_{P,I}$, 83
$\lceil$, 28
$\vdash_P$, 10, 12
c-micro-limited, 49

abstract parse tree, 24
Acc($\mathcal{A}$), 10, 13
accept, 10, 13
active, 9, 12
active domain, 6
active structure, 9, 12
ActiveOrder$_{\Upsilon}$, 27
alternating ASM, 84
APT, 24
ASM, 9
ASPACE$_i(f(n))$, 85
ATIME$_i^{\mathrm{macro}}(f(n))$, 85
ATIME$_i^{\mathrm{micro}}(f(n))$, 85
atomic, 7
atomic formula, 51

backgound structure, 9
background, 9, 12
backgroundstructure, 12
BCVP$_{\Upsilon}$, 60
block, 7
Blum-Shub-Smale-machine, 112
body, 7, 11
Boole, 6
Boolean expession, 62
Boolean-valued term, 5
BSS-machine, 112

circuit, 59
clause, 7
complete, 23
complexity class, 23
computation graph, 20
computation tree, 83
concrete parse tree, 24
conditional, 7
configuration, 98
consistent, 8, 11
content, 8
contradictory, 8, 11
converging, 10, 12
Cook's Theorem, 120
copy-solvent, 27
CPT, 24

deterministic rule, 8
deterministic update set, 8
diagonalization, 26
diagonalization theorem, 26
diagonalization-solvent, 25
disabling, 10, 12
diverging, 10, 12

element-non-determinism, 12
EmptyCopy$_{\Upsilon}$, 27
EmptyGraph$_{\Upsilon}$, 27
enabling, 10, 12
encode, 25
equivalent ASMs, 14
equivalent rules, 14
existential second-order sentence, 52
Exp, 6
expanded state, 6
expansion, 6

false, 6
final, 10, 12
fire, 8
first-order interpretation, 59
$\mathrm{FO}_{\mathrm{static}}$, 51
$\mathrm{FO}_{\mathrm{static}}$-reducible, 59
$\mathrm{FO}_{\mathrm{static}}$-reduction, 60

graph-solvent, 27
ground, 5
guard, 7, 11
guess-element, 11, 13
guess-rule, 11, 13

hang, 10
hanging, 10, 12
hard, 23
head variable, 11
HET, 30
honest evaluation time, 30
Horn-formula, 67
HORN-SAT, 67

import, 7
init, 29
initial, 23
Inputstates(P), 12

Linear Time Hierarchy Theorem, 40
living, 6
location, 8
lock step simulation, 97
logspace-reduction, 68

macro time, 30
macro time constructible, 38
macrostep, 30
micro time, 31
micro time constructible, 40
microstep, 30

$\mathrm{NHET}_{\mathrm{el}}$, 32
$\mathrm{NHET}_{\mathrm{rule}}$, 32
non-deterministic ASM, 12
non-deterministic BSS-machine, 118
non-deterministic honest evaluation time, 32
non-deterministic macro time, 31
non-deterministic RAM, 111
non-deterministic rule, 11
non-deterministic update set, 11
$\mathrm{NP}_{\mathrm{el},\Upsilon}$, 60
$\mathrm{NP}_{\mathrm{el},\Upsilon}$-hardness, 60
$\mathrm{NP}_{\mathrm{el}}$, 52
$\mathrm{NP}_{\mathrm{rule}}$-completeness, 62
$\mathrm{NP}_{i,\Upsilon}$, 50
NP_{i}, 50
NPSPACE, 75
$\mathrm{NSPACE}_{i,\Upsilon}(f(n))$, 74
$\mathrm{NSPACE}_{i}(f(n))$, 74
$\mathrm{NSPACE}_{i}(P, I)$, 74
$\mathrm{NSPACE}_{i}(P, L, n)$, 74
$\mathrm{NTIME}_{i}^{\mathrm{macro}}(f(n))$, 33
$\mathrm{NTIME}_{i}^{\mathrm{macro}}(P, L, n)$, 32
$\mathrm{NTIME}_{i}^{\mathrm{micro}}(f(n))$, 40
$\mathrm{NTIME}_{i}^{\mathrm{micro}}(P, I)$, 33
$\mathrm{NTIME}_{i}^{\mathrm{micro}}(P, L, n)$, 33

order-solvent, 27

P, 50
P_{Υ}, 50
P-completeness, 68
parse tree, 23
partial run, 10, 12
PASM, 92
polynomial, 112
polynomial macro-time-reduction, 62
probabilistic ASM, 92
progam, 12
program, 9
program over Υ, 9
PSPACE, 75

RAM, 100
rational, 112
reachable, 10, 12
real input, 9
reduction, 23

reject, 10
relation, 5
relational, 5
reserve, 6
rule, 8, 11
rule over Υ, 9
run, 10, 12

SAT, 62
size, 21
space constructible, 71
space hierarchy, 75, 76
SPACE$(f(n))$, 71
SPACE(P, I), 71
SPACE(P, L, n), 71
SPACE$_\Upsilon(f(n))$, 71
Speed-Up-Theorem, 33, 40, 119
stat-Σ^1_1, 52
state, 5, 12
States(P), 9, 12
static, 5
strict lock step simulation, 97
structure of finite size, 21
successor, 10, 12

term, 5
time hierarchy, 41–43
TIME$^{\text{macro}}(f(n))$, 33
TIME$^{\text{macro}}(P, I)$, 30
TIME$^{\text{macro}}(P, L, n)$, 30
TIME$^{\text{macro}}_\Upsilon(f(n))$, 33
TIME$^{\text{micro}}(f(n))$, 39
TIME$^{\text{micro}}(P, I)$, 31
TIME$^{\text{micro}}(P, L, n)$, 31
TIME$^{\text{micro}}_\Upsilon(f(n))$, 39
true, 6
Turing machine, 98

undef, 6
update, 8
update set, 8, 11

variable assignment, 6
voc, 5
vocabulary, 5
vocabulary of a rule, 9