Abstract State Machines: Verification Problems and Computational Power

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades einer Doktorin der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatikerin

Antje Nowack

aus Düsseldorf

Berichter: Universitätsprofessor Dr. Erich Grädel
Universitätsprofessor Dr. Wolfgang Thomas

Tag der mündlichen Prüfung: 21. Juli 2004

Zusammenfassung

Abstract State Machines (ASMs) bilden die Basis einer Methode zur Software-Spezifikation, welche Vorteile informaler Methoden (Verständlichkeit und Ausführbarkeit der Spezifikation) mit Vorteilen formaler Methoden (Genauigkeit sowie Anwendbarkeit mathematischer Methoden und Ergebnisse) vereint. Anwendungen dieser Methode motivieren zahlreiche Berechnungs- und Entscheidungsprobleme. Die hohe Ausdrucksstärke ist einer der Vorteile der ASMs, jedoch führt sie fast unmittelbar zu Unentscheidbarkeits- bzw. Nichtberechenbarkeitsergebnissen im uneingeschränkten Fall. Folglich gelangt man recht schnell zu der Frage, ob es ausdrucksstarke Klassen von ASMs gibt, für welche Enscheidbarkeits- und Berechenbarkeitsergebnisse bewiesen werden können. Im ersten Teil dieser Arbeit wird eine solche Klasse eingeführt. Diese ASMs werden bewachte ASMs genannt. Die Idee ähnelt derjenigen des bewachten Fragments der Logik erster Stufe, für welches Erfüllbarkeit entscheidbar ist. Weiterhin wird die Ausdrucksstärke dieser Klasse analysiert und bewiesen, daß sie ausdrucksstärker ist als Datalog LITE und als das bewachte Fragment der Fixpunktlogik erster Stufe.

Im zweiten Teil der Arbeit wird die Entscheidbarkeit des allgemeiner Verifikationsprobleme betrachtet. Dies entspricht der Frage, ob alle Berechnungen einer ASM eine Eigenschaft erfüllen. Aufgrund der Unentscheidbarkeit im allgemeinen Fall, müssen die ASMs und die Eigenschaften eingeschränkt werden, um Entscheidbarkeit zu erhalten. Bewachte ASMs bilden die Basis einer entscheidbaren Instanz des allgemeinen Verifikationsproblems.

An diese Betrachtungen schließt sich die Frage an nach der Möglichkeit, die Beschränkungen der ASMs abzuschwächen, falls das Ziel nicht in der automatischen Verifikation sondern in Konzepten besteht, die Verifikation, Debugging, Testen unterstützen. Eine solches Konzept ist Slicing für ASMs, welches im dritten Teil dieser Arbeit eingeführt wird. Die Idee entspricht derjenigen des Program Slicing, dessen Ziel ist, diejenigen Anweisungen eines Programms zu bestimmen, welche sein Verhalten an einer gegebenen Stelle beeinflussen. Diese Anweisungen bilden wiederum ein syntaktisch korrektes Programm, welches Slice genannt wird. Bisherige Arbeiten haben Programmiersprachen betrachtet, deren Konzept sich grundsätzlich von ASMs unterscheidet. Obwohl das Konzept des Program Slicing nicht direkt auf ASMs erweitert werden kann, läßt sich ein entsprechendes Konzept für ASMs finden. Ein solcher Ansatz wird hier vorgestellt. Obwohl ein minimaler Slice im Allgemeinen nicht berechenbar ist, wird bewiesen, daß ein minimaler Slice für bewachte ASMs berechenbar ist. Dieses Ergebnis kann auf mehrere Arten erweitert werden. Einige Erweiterungen auf größere Klassen von ASMs sowie weitere Varianten des Slicing-Begriffs werden vorgestellt.

Im vierten Teil dieser Arbeit wird die Betrachtungsweise der ASMs etwas verändert. ASMs werden nicht nur als Spezifikationsformalismus gesehen sondern als Berechnungsmodell. Die ASM-These besagt, daß jeder Algorithmus, gleich welcher Art, schrittweise und auf seinem natürlichen Abstraktionsniveau durch eine ASM simuliert werden kann. Diese These wurde für sequentielle und parallele synchrone Algorithmen von grundlegenden Prinzipien abgeleitet. Das Hauptergebnis dieses Teils ist, daß die ASM-These auch für Quanten-Algorithmen gilt.

Abstract

Abstract State Machines (ASMs) provide the basis of a a formal method combining advantages of informal methods (understandability, executability) and advantages of formal methods (precision and applicability of mathematical methods and results). Applications of this method motivate numerous computability and decidability problems. The high expressive power is one of the advantages of ASMs but it leads rather directly to undecidability respectively uncomputability results in the unrestricted case. Consequently, we arrive rather early at the question whether there exist expressive classes of ASMs for which we can prove decidability and computability results. In the first part of this thesis, we introduce such a class called guarded ASMs. The idea is similar to the one of the guarded fragment of first-order logic for which satisfiability is decidable. We analyze the expressive power of this class and prove that it is (strictly) stronger than Datalog LITE and the guarded fragment of first-order fixed point logic.

In the second part of this thesis, we study the decidability of general verification problems for ASMs corresponding to the question whether all computations of an ASM satisfy a property (usually expressed in some temporal logic). Because of undecidability in the general case, we have to restrict the ASMs and the properties in order to obtain decidability results. Guarded ASMs provide the basis of a decidable instance of the general verification problem.

It is rather straightforward to ask for the possibility to weaken the restrictions on the ASMs if we do not aim automatic verification but concepts supporting verification, debugging and testing. One such possibility is the concept of slicing ASMS which we introduce in the third part of this work. The idea is analogous to the one of program slicing aiming to extract statements from a program that are relevant for its behavior at a given point of interest. These statements form again a syntactically correct program called a slice. Previous work has focused on programming languages that differ substantially from ASMs. Although the concept of program slicing does not directly extend to ASMs, it is possible to find an analogous concept for ASMs. We present such an approach. In spite of the fact that a minimal slice is not computable in the general case, we prove that a minimal slice is computable for guarded ASMs. This basic result can be extended in several ways. We present some extensions to larger classes of ASMs and other variants for the notion of slicing.

In the fourth part of this thesis, we change our point of view. We do not merely consider ASMs as a specification formalism but as a computation model. The ASM thesis says that every algorithm, of any kind, can be modeled step by step and on its natural abstraction level by an ASM. The thesis has been derived from basic principles for sequential algorithms, and for parallel synchronous algorithms. The main result of this part is that the ASM thesis also holds for quantum algorithms.

Acknowledgements

I want to thank all people who contributed directly or indirectly to this thesis. In particular, I want to thank my advisor, Professor Erich Grädel, for his support and guidance during the last years. Furthermore, I want to thank Professor Wolfgang Thomas for his kind readiness to write the second report on this thesis.

Contents

Introduction

Abstract State Machines (ASMs) (formerly known as Evolving Algebras) have been introduced to bridge the gap between formal models of computation and practical specification methods. The result is a formal method for transparent design and specification of discrete dynamic systems. ASMs combine advantages of informal methods (understandability, executability) with the advantages of formal methods (precision and applicability of mathematical methods and results).

On the one hand, the ASM method is a formal method. It is based on a precise semantics allowing the application of mathematical methods to analyze ASMs. In particular, results from mathematical logic are often exploited as ASMs use classical mathematical structures to describe states of a computation.

Nevertheless, ASMs are easily understandable. ASM specifications are easy to read and to write. This understandability is one of the most important preconditions for real applications and can be partially ascribed to rather strong constructs causing that, for example, an encoding of inputs is not necessary and algorithms can be formulated in an intuitive way. The syntax of ASMs is quite similar to pseudo-code and the resulting computation model is very powerful. This manifests in the ASM thesis. It says that *"every algorithm of any kind is modeled, step by step and on its natural abstraction level, by an abstract state machine"*. There is enough evidence for this proposition resulting from numerous examples and the derivation for sequential and parallel algorithms from basic postulates (see [21] and [5]). Caused by a lack of a formal definition of the notion of algorithm, there is no proof in the general case.

A further important property is that ASM specifications are executable. The executability is an immediate consequence of the conception of ASMs and strongly supports their application in the area of software development. There are tools for executing ASMs such as AsmL (abbreviating Abstract State Machine Language) developed by Microsoft Research or the ASM Gofer. They provide the possibility to write, execute and test ASM specifications so that errors can be found and repaired at a very early stage of the design process.

In addition to the preceding features, successful applications of the ASM method substantiate that this approach indeed bridges the gap between formal models of computation and practical specification methods. Therefore, the aim that has been the starting point of the ASM project is reached. Examples of applications

can be found in [6] and [33].

We have already hinted that ASMs can be considered as a computation model as well as a specification method. In each of these cases, we are concerned with different problems.
In the case that we consider ASMs as a specification method, we are rather early concerned with the task of finding errors in a specification as writing formal models is not an error-free process. There are two basic approaches to detect errors.

The first approach is to verify properties automatically. It is clear that this is not possible in many cases as the question whether an ASM satisfies a property is undecidable in the unrestricted version. In Part II, we consider automatic verifiability of ASMs in the case that we put restrictions on the ASMs and the admitted properties. The main restriction is introduced in Part I. It results from the transfer of the restrictions of an expressive decidable instance of the satisfiability problem for first-order logic to ASMs. We analyze the computational power of the corresponding classes of ASMs mainly via comparisons to other well-known formalisms.

The second approach is to test ASMs. We have already been mentioned that the executability yields this possibility. We do not investigate testing itself but we consider a technique that might be combined with testing (but can also be combined with verification). This technique is called slicing and is considered in Part III of this thesis. Essentially, we cut pieces from the ASM and put them together to a new ASM. The pieces are chosen in such a way that the new ASM behaves in the same way as the original one if we consider only a specified part of the states. This can be compared to the limited view through a window. E.g., if an error is observed we are only interested in that part of the ASM that is responsible for the error. The resulting ASM might be much smaller than the original ASM. Therefore, it becomes a much easier task to locate and correct the error.

Note that testing ASMs has also been investigated in another direction. For example, the AsmL Test Generator tool developed by Microsoft Research provides the possibility to generate test cases for an implementation from its ASM specification. A similar approach has been considered in [11]. It would also be advantageous to combine slicing with these approaches or with semi-automatic approaches as the translation to interactive provers as PVS (see e.g. [9], [10]).

If we change our point of view and consider ASMs as a computation model then we are faced with other kinds of questions. One such question is whether the ASM thesis really holds for *all* kinds of algorithms. Though there is enough evidence for this thesis, we do not have a proof in the general case. For sequential (see [21]) and parallel algorithms (see [5]), the ASM thesis has been derived from basic postulates. A question that has been asked repeatedly is whether the ASM thesis

holds also for quantum algorithms. We answer this question in Part IV of this thesis.

In the remainder of the introduction we give a short overview on each of the four parts.

Part I. Guarded Abstract State Machines

In the first part of this thesis, we introduce a restriction on ASMs similar to the one on formulae in the guarded fragment of first-order logic (GF). GF originates from an investigation of the reasons for the good model-theoretic properties of propositional modal logic (ML). It is possible to embed ML into first-order logic and translate every formula from ML to first-order logic. In [3], H. Andréka, J. van Benthem and I. Németi have considered this translation (leading to the modal fragment) a bit closer and detected that in the modal fragment, all quantifications are guarded by some atomic formula. This observation then leads to the definition of the guarded fragment of first-order logic.

It is easy to see that the problems addressed in the second and in the third part of this thesis are strongly connected to the satisfiability problem for first-order logic. Further, it is known that the (finite) satisfiability problem for GF is decidable.

In this part, we focus on the class that results from applying the restriction to the class of deterministic ASMs. We investigate its computational power, mainly via comparisons to other well-known formalisms such as Datalog LITE or guarded first-order fixed-point logic. As a rough result of this part we obtain that despite the restrictions this class of ASMs has high computational power. It is even stronger than the formalisms used in the comparisons.

Part II. Verification

The aim of verifying ASMs is to prove that the behavior of an ASM specification coincides with the desired one. In contrast to testing, one covers *all* possible cases and not just a (finite) part.

The ideal case would be to have an algorithm deciding, for a given ASM and a given property (typically formulated in some temporal logic), whether the ASM satisfies the property. There are mainly two possible meanings for the question whether "the behavior of an ASM specification coincides with the desired one":

1. Do all computations of an ASM satisfy a property? (general verification problem)

2. Do the computations of an ASM with a given initial structure satisfy a property? (restricted verification problem)

As we have already mentioned, ASMs are very powerful. It is easy to prove that both variants of the verification problem are undecidable in the general case. In this part, we introduce some instances of the verification problem and investigate the decidability of the verification problem for these instances.

The focus of this part is on a well-defined fragment of the class introduced in the first part of this thesis. We prove that the general verification problem is decidable for this fragment and an expressive fragment of linear temporal first-order logic.

A first idea might be to exploit results from the area of model checking in order to obtain results on the verification of ASMs. Model checking is an algorithmic method for checking whether a system satisfies a specification. The system is modeled by a structure (e.g. a transition system) and the specification is a formula in an appropriate logic. Then one checks whether the structure is a model of the formula.

Meanwhile, model checking has become an established industrial practice. Model checking algorithms for finite state systems proceed by a systematic examination of the entire state space. In most cases, it is not possible to exploit this approach for the verification of ASMs as one deals with infinite state spaces. Consequently, we have to look for other ways to carry out model checking.

Part III. Slicing

In order to debug or to modify a program, we are most often not interested in its complete behavior but only in its behavior at a given point of interest. This means that we do not observe each complete program state but only a specified part of it. Consider for instance the case of debugging. When an error is observed, the programmer tries to extract that part of the program which is responsible for the erroneous behavior. This set of statements might be much smaller than the original one and therefore, it might be much easier to catch and correct the error(s) if he knows this set.

The above observation motivates the concept of program slicing. By applying this established technique, one extracts statements from a program that are relevant for the behavior of the program at some specified point of interest. The result is a program slice which forms a syntactically correct program again.

Program slicing has first been considered by M. Weiser [37]. However, his considerations are only based on block-structured, possibly recursive programs written in a Pascal-like language. Weiser defined the notion of slice as follows. Let P be a program, p be the label (e.g. an index) of a statement in P and V be a subset of the program variables of P. We are interested in the (sequence of) values assigned to V just before the statement with label p is executed. A (static) slice of P relative to the slicing criterion $\langle p, V \rangle$ is obtained from P by removing statements in such a way that the values of the variables in V just before

executing the statement with label p are the same for the complete program P and its slice.

As every program is a slice of itself, it is clear that a slice does always exist. Furthermore, there are usually many slices of a program. One can say that the smaller a slice the better. Therefore, it would be ideal to have a minimal slice of a program. Unfortunately, minimal slices are not automatically computable in general.

There has been extensive research on program slicing, on automatic computation of slices, and on methods for approximating minimal slices. See [34] for an overview.

ASMs are a formal model of computation as well as an executable specification language used in practice. This suggests to combine practically established concepts with theoretically founded approaches for validating ASM specifications.

One technique widely spread in practice is to validate programs via testing. The executability implies that testing can also be applied to ASMs. Testing becomes more efficient and more accurate if it is appropriately combined with theoretically sound concepts. In this sense, the concept of slicing is a good supplement for testing assumed that we have a formal basis for slicing. Another approach that could be combined with slicing is to generate test cases from an ASM specification in order to check an implementation against its specification on these test cases. This has been realized in the AsmL Test Generator tool developed by Microsoft Research. Another approach has been presented in [11]. Moreover, there are many other fields where a combination with slicing has an advantageous effect. For example, it would be reasonable to combine it with verification (regardless whether manual, semi-automatic, automatic) or to use it for modifying existing ASMs.

The contribution of this part is as follows.

Formal introduction of slicing for ASMs. The design of the programming languages that are originally considered for slicing differs substantially from the one of ASMs such that the original concept of program slicing does not extend directly to ASMs. In spite of these differences, it is possible to find a concept for ASMs that is analogous to the one of program slicing. We present such an approach.

A computability result. It is easy to see that a minimal (static) slice cannot be computed automatically for unrestricted ASMs. However, we present a class of ASMs for which a minimal slice is computable (and prove the computability). This is the class that we investigated in Part I of this thesis.

Extensions. Once we have proven the basic computability result it is easy to obtain several extensions of it. First, we extend the class and explain how

the basic result extends to these super-classes. Afterwards we explain how one can vary the notion of slice and how the basic result can be adapted to these variations.

Equivalence of ASMs. After reasoning about slicing, we consider an approach to decide the equivalence of ASMs (taken from some class of ASMs). In a certain sense, the question of equivalence of two ASMs is closely related to slicing of ASMs. A slice of an ASM is partially equivalent to the ASM itself. I.e., it behaves equivalently to the original ASM on a specified set of locations.

Compared to computing a minimal slice, deciding the equivalence of two ASMs is rather straightforward.

Part IV. Quantum Computing

As previously mentioned, we change our point of view in this part of the thesis. We do not consider ASMs merely as a specification formalism but as a model for computation on abstract structures, which is the basis of a successful methodology for specification of hardware and software systems, with practical applications in industrial environments. In this way, ASMs become also interesting for the theory of computation, for instance for the very fundamental question of what constitutes an algorithm. The connection between ASMs and algorithms is mainly reflected in the ASM thesis. There is considerable empirical evidence for this thesis, and in important cases, the thesis can actually be derived from basic postulates.

In [21], Gurevich showed that this is the case for *sequential algorithms*. He formulated three basic postulates for sequential algorithms and proved that every algorithm satisfying these three postulates can be simulated, step by step, by a sequential ASM. It should be stressed that this notion of simulation or, in other words, the equivalence of two algorithms, is meant in a much more precise way than, for instance, in the Church-Turing thesis where simulation just means to have the same input-output-behavior (to compute the same function). Here one requires complete behavioral equivalence: the two algorithms have the same states and initial states, the same one-step transformation, and produce at every step the same output.

Later, Blass and Gurevich [5] extended that work to *parallel algorithms* which still work in sequential time, i.e. in discrete time steps governed by a global clock but with unbounded parallelism in each time step. For the more general model of distributed, asynchronous algorithms, there is so far no such treatment, but work seems to be under way.

However, it has been repeatedly asked, and so far not been answered in a satisfactory way, whether the ASM thesis also holds for nonstandard models of algorithms such as *quantum algorithms*. One of the reasons, why this question

has not really been looked into, may be the widespread view in academia and industry that quantum computing provides a purely theoretical playground for mathematicians and physicists rather than "real algorithms". Indeed the main theoretical success of quantum computing is the polynomial-time quantum factorization algorithm by Shor [29], but on the practical side there are enormous difficulties for building quantum computers with more than a few qubits. The recent news that scientists at IBM have successfully factored the number 15 using a quantum computer with seven qubits [36] has probably not shaken that view. Still, the question remains, and provides a challenge for the advocates of the ASM thesis. First of all, to our present knowledge, the laws of physics seem not to exclude that large quantum computers can eventually be built, so the present state of quantum computers may just reflect our technological maturity rather than the potential of quantum computing. Second, quantum computing defines a mathematical model of computation (in fact: several models), and a thesis that comes with the ambition of shedding light on the general notion of algorithm should take care of this model as well.

Thus, we felt that the ASM thesis should apply also to quantum algorithms, and our intuition told us that it does. The purpose of this part is to explain that and how ASMs can model quantum algorithms.

Basic Definitions of Abstract State Machines

The basic idea of (deterministic) Abstract State Machines (ASMs) is the following. The states of an ASM are countable first-order structures where only a finite subset of the domain occurs in the functions and relations. This finite part of a state is called the active domain of the state. The remainder of the domain is referred to as the reserve.

An ASM is defined via a program determining the transitions between the states. The basic components of programs are update rules of the form $u := v$ where u and v are terms over a given vocabulary. Further types of rules have the form **if** φ **then** R **endif** and **forall** $\overline{x}$: φ **do** R **endforall** where R is a rule and φ a first-order formula. Their semantics is the intuitive one. The rule inside an if-clause is executed if, and only if, the guard φ evaluates to true. Note that quantifiers and forall-rules have to be read as relativized to the active domain. The inner rule of a forall-rule is executed in parallel for all tuples of elements satisfying the guard φ of the forall-rule. **import** allows to address an element from the reserve and therefore, such an element can be inserted into the active domain. A program is a sequence of such rules. In one step of the program, they are all executed in parallel. The program is executed repeatedly.

Note that in the above informal explanation and in the subsequent formal definition, the notion of parallel execution of rules or updates is meant in the sense of real parallelism. The rules or updates are executed simultaneously and this is *not* meant e.g. in the sense of arbitrary interleaving. The updates happen at exactly the same time. And this is the reason why two updates or rules might contradict each other. If e.g. the same item should be set to true and to false at the same time, then this is a contradiction and can not be carried out in a canonical way. We will later address this problem again and present the solution chosen in the context of ASM.

In this chapter, we give only a short introduction to ASMs. For detailed information on ASMs (formerly called evolving algebras) see e.g. , [7], [19], [20].

Vocabulary. An ASM-vocabulary Υ is a finite set of function symbols. Υ contains at least the constant symbols Mode, initial, undefined, true, false and $\vee/2$, $\wedge/2$, $\neg/1$.

Relations can be considered as special functions marked relational and the application of such a function can only lead true or false.

In the following, the notion relation (symbol) will refer to those function (symbols) marked relational and the notion function symbol to those not marked relational.

Furthermore, a boolean-valued term is a term whose outermost symbol is a relation symbol and therefore, the term has to be evaluated to true or to false.

Active domain. The active domain of an Υ-structure $\mathcal{A}$ is a subset of the domain of $\mathcal{A}$ not containing undefined such that for every element x of the active domain the following holds:

- there are an n-ary function symbol $f \in \Upsilon$ and elements a_1, ..., a_n of the domain of $\mathcal{A}$ such that $f^{\mathcal{A}}(a_1, ..., a_n) = x$ or $f^{\mathcal{A}}(a_1, ..., a_j, x, a_{j+1}, ...a_{n-1}) = a \neq$ undefined or
- there are an n-ary function $R \in \Upsilon$ and elements a_1, ..., a_{n-1} of the domain of $\mathcal{A}$ such that $R^{\mathcal{A}}a_1...a_j x a_{j+1}...a_{n-1} =$ true.

For a structure $\mathcal{A}$, let $\mathrm{ad}(\mathcal{A})$ denote the active domain of $\mathcal{A}$.

Reserve. The reserve of an Υ-structure $\mathcal{A}$ is the set of all elements in the domain of $\mathcal{A}$ that are not in the active domain of $\mathcal{A}$. (In other words, the reserve is a naked set except that equality resp. inequality are defined on it.)

For a structure $\mathcal{A}$, let $\mathrm{reserve}(\mathcal{A})$ denote the reserve of $\mathcal{A}$.

State. A state $\mathcal{A}$ of vocabulary Υ is a countable first-order Υ-structure with a finite active domain.

Input and Initial State. An input for an ASM over vocabulary Υ is a finite first-order structure $\mathcal{A}$ over $\Upsilon - \{\mathrm{Mode}, \mathrm{initial}, \mathrm{undefined}, \mathrm{true}, \mathrm{false}, \vee, \wedge, \neg\}$. It is mapped to an initial structure $\mathcal{I}$ as follows:

1. $\mathrm{ad}(\mathcal{I}) = \mathrm{ad}(\mathcal{A})$
2. $\mathcal{I}|_{\mathrm{ad}(\mathcal{A})} = \mathcal{A}$
3. $\mathrm{Mode}^{\mathcal{I}} = \mathrm{true}$
4. the reserve of $\mathcal{I}$ is a countable (infinite) naked set (disjoint from $\mathrm{ad}(\mathcal{A})$.
5. $\mathrm{initial}, \mathrm{undefined}, \mathrm{true}$ and false are interpeted by distinct elements.
6. $\wedge$, $\vee$, $\neg$ are only defined on $\{\mathrm{true}, \mathrm{false}\}$. On this set, they are interpreted as usual.

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

Syntax of ASMs Let Υ be a finite vocabulary (containing only relation and constant symbols). ASM rules over Υ are defined inductively:

Skip Skip is an ASM rule.

Update rule For all terms s (possibly boolean-valued) over Υ and all terms t over Υ, the assignment $s := t$ is a rule. If s is a boolean-valued term then t has also to boolean-valued.

Conditional rule If g is an FO-formula over the vocabulary Υ and R' is an ASM rule (over the vocabulary Υ) then **if** g **then** R' **endif** is an ASM rule (with guard g).

Parallel composition If R_1 and R_2 are ASM rules then their parallel composition **do-in-parallel** R_1 R_2 **enddo** is an ASM rule.

Parallelism Let $\overline{z}$ be a tuple of variables, R' an ASM rule, φ an FO-formula over the vocabulary Υ. Then **forall** $\overline{z} : \varphi$ R' **endforall** is a rule.

Import If v is a variable and R an ASM rule then **import** v R' **endimport** is an rule.

Element-nondeterminism Let $\overline{z}$ be a tuple of variables, R' an ASM rule, α an FO-formula over the vocabulary Υ. Then **choose** $\overline{z} : \alpha$ R' **endchoose** is an ASM rule.

Rule-nondeterminism Let R_1 and R_2 be ASM rules. **choose** R_1 R_2 **endchoose** is an ASM rule in this case.

Free and bounded variables are defined as usual with **forall** $\overline{x} : \varphi$, **choose** $\overline{x} : \varphi$ binding $\overline{x}$ and **import** v binding v. A program is an ASM rule without free variables.

Every ASM program Π defines an ASM. In the remainder of this work, we identify the program Π with the ASM defined by Π. There are also other notions where an ASM is not merely defined by a program but also by an initialization mapping, a formula defining its input states and a vocabulary. An initialization mapping maps inputs to initial states. Here, if nothing conversely is explicitly stated, the vocabulary is induced by the program, the initialization is defined as already described and any state over the vocabulary of the ASM is allowed to be an input.

In some proofs, we use the notion of intended interpretation. It corresponds to the idea of initialization mapping. Sometimes, we are only interested in a part of the possible initial states. This is for example the case, if we consider a translation from another formalism to ASMs. Assume that we consider a computation model with words as inputs. In order to obtain an appropriate input for an ASM, we have to map words to structures. Consequently, we do not obtain arbitrary initial states of ASMs but a class with certain restrictions. Another possibility is that we want a relation to be initially empty as we accumulate tuples during a computation. We formulate all such restrictions in intended interpretations.

Location and Content. A slight change in the way of considering states simplifies the later investigations. A state $\mathcal{A}$ can be considered as a map from locations to contents. Locations have the form $(f, \overline{a})$ where f is a function or a relation symbol of arity r and $\overline{a}$ is an r-tuple of elements of the domain. The content of $(f, \overline{a})$ is $f^{\mathcal{A}}(\overline{a})$.

Relations are considered as special functions which can only have true or false as values. Consequently, the content of location with a relation symbol can only be true or false.

Update. An update of a state $\mathcal{A}$ is a pair $\beta = (l, a)$ where l is a location over $\mathcal{A}$ and a an element from the domain of the structure. Executing this update means to put a into l (i.e., set the content of l to a) and leave the other locations as they are.

Update Set. A deterministic update set is a set of updates. A nondeterministic update set is a set of deterministic update sets.

Contradictory and Consistent. A deterministic update set U is contradictory if it contains two updates (l, a) and (l, b) with $a \neq b$. Otherwise, U is consistent.

A nondeterministic update set is contradictory if all its deterministic elements are contradictory. Otherwise, it is consistent.

Executing Update Sets. To execute a deterministic update set, do nothing; the new state is identical to the old one. To execute a consistent deterministic update set $\{(l_1, a_1), \ldots, (l_k, a_k)\}$, put the elements $a_1, \ldots, a_k$ into the locations $l_1, \ldots, l_2$ respectively and leave the other locations as they are.

To execute a consistent nondeterministic update set, execute one of its consistent deterministic elements.

To execute a contradictory update set, do nothing.

Semantics of ASMs. The evaluation of a deterministic rule R at a state $\mathcal{A}$ under a variable assignment ξ produces a deterministic update set $U = \Delta(R, \mathcal{A}, \xi)$. To execute R, execute U; the result is the successor of $(\mathcal{A}, \xi)$ with respect to R. R is consistent (respectively contradictory) at $(\mathcal{A}, \xi)$ if U is consistent (respectively contradictory) .

The deterministic update set produced by a deterministic rule R is defined by induction on R:

- $\Delta(\text{Skip}, \mathcal{A}, \xi) = \emptyset$
- Given an update rule of the form $f(t_1, ..., t_r) := t_0$ and an expanded state $(\mathcal{A}, \xi)$, evaluate all terms t_i. Let a_i be the resulting value of t_i, $\overline{a} = (a_1, ..., a_r)$ and l the location $(f, \overline{a})$. Then $\Delta(R, A, \xi) = \{(l, a_0)\}$.
- Given a conditional rule as described in the definition of the ASM syntax and an expanded state $(\mathcal{A}, \xi)$, evaluate the guard. The semantics for the guards is the same as for first-order logic except that the guards range only over the active domain of the structure.

  If the guard evaluates to false, the $\Delta(R, \mathcal{A}, \xi) = \emptyset$. If g evaluates to true, then $\Delta(R, \mathcal{A}, \xi) = \Delta(R', \mathcal{A}, \xi)$.
- Given a parallel composition as described in the definition of the ASM syntax and an expanded state $(\mathcal{A}, \xi)$, evaluate R_1 and R_2. $\Delta(R, \mathcal{A}, \xi) = \Delta(R_1, \mathcal{A}, \xi) \cup \Delta(R_2, \mathcal{A}, \xi)$
- Given a forall-rule as described in the definition of the ASM syntax and an expanded state $(\mathcal{A}, \xi)$. Let $M := \{\overline{a} \in \text{domain}(\mathcal{A})^r : a_1, ..., a_r$ are non-reserve and $\mathcal{A} \models \varphi[\overline{z}/\overline{a}]\}$.

  $\Delta(R, \mathcal{A}, \xi) = \bigcup_{\overline{a} \in M} \Delta(R', \mathcal{A}, \xi[\overline{z} \to \overline{a}])$ where $\xi[\overline{z} \to \overline{a}]$ is the variable assignment with domain $V' = V \cup \{z_1, \ldots, z_r\}$ obtained from ξ by setting or resetting $\xi[\overline{z} \to \overline{a}](z_i) = a_i, i \in \{1, \ldots, r\}$.
- Given an import rule as described in the definition of the ASM syntax and an expanded state $(\mathcal{A}, \xi)$ where ξ has the 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' at $(A, \xi[v \to a])$. $\Delta(R, \mathcal{A}, \xi) = \Delta(R', \mathcal{A}, \xi[v \to a])$.

  Two elements imported at the same time are not allowed to be equal.

The evaluation of a nondeterministic rule R at a state $\mathcal{A}$ and a variable assignment ξ produces a nondeterministic update set $U = \Delta^{\text{N}}(R, \mathcal{A}, \xi)$. To execute R, execute U; the result is a successor of $(\mathcal{A}, \xi)$ with respect to R. R is consistent (resp. contradictory) if U is consistent (resp. contradictory).

The nondeterministic update set produced by a nondeterministic rule R is defined by induction on R.

- $\Delta^{\mathrm{N}}(\mathrm{Skip}, \mathcal{A}, \xi) = \{\emptyset\}$

- Given an atomic rule of the form $f(t_1, ..., t_r) := t_0$ and an expanded state $(\mathcal{A}, \xi)$, evaluate all terms t_i. Let a_i be the resulting value of t_i, $\overline{a} = (a_1, ..., a_r)$ and l the location $(f, \overline{a})$. Then $\Delta^{\mathrm{N}}(R, \mathcal{A}, \xi) = \{\{(l, a_0)\}\}$.

- Given a conditional rule as described in the definition of the syntax of ASMs and an expanded state $(\mathcal{A}, \xi)$, evaluate the guard. The semantics for the guards is the same as for first-order logic except that the guards range only over the active domain of the structure. If the guard evaluates to false, then $\Delta^{\mathrm{N}}(R, \mathcal{A}, \xi) = \{\emptyset\}$. If g evaluates to true, then $\Delta^{\mathrm{N}}(R, \mathcal{A}, \xi) = \Delta^{\mathrm{N}}(R', \mathcal{A}, \xi)$.

- Given a parallel composition as described in the definition of the syntax of ASMs and an expanded state $(\mathcal{A}, \xi)$ then $\Delta^{\mathrm{N}}(R, \mathcal{A}, \xi) = \{X_1 \cup X_2 : X_1 \in \Delta^{\mathrm{N}}(R_1, \mathcal{A}, \xi), X_2 \in \Delta^{\mathrm{N}}(R_2, \mathcal{A}, \xi)\}$.

- Given a forall-rule as described in the definition of the syntax of ASMs and an expanded state $(\mathcal{A}, \xi)$. Let $M := \{\overline{a} \in \mathrm{domain}(\mathcal{A})^r : a_1, ..., a_r$ are non-reserve and $\mathcal{A} \models \varphi[\overline{z}/\overline{a}]\}$.

  $\Delta^{\mathrm{N}}(R, \mathcal{A}, \xi) = \{\bigcup_{\overline{a} \in M} X_{\overline{a}} : X_{\overline{a}} \in \Delta^{\mathrm{N}}(R_0, \mathcal{A}, \xi[\overline{z} \to \overline{a}])\}$ where $\xi[\overline{z} \to \overline{a}]$ is obtained as in the deterministic case.

- Given an import rule as described in the definition of the syntax of ASMs, a state $\mathcal{A}$ and a variable assignment ξ with domain V, let $V' = V \cup \{v\}$ (where v is an element of the reserve) and $\xi[v \to a]$ is obtained as in the deterministic case. Evaluate R' at $(\mathcal{A}, \xi[v \to a])$. $\Delta^{\mathrm{N}}(R, \mathcal{A}, \xi) = \Delta^{\mathrm{N}}(R', \mathcal{A}, \xi[v \to a])$.

  As in the deterministic case, two elements imported at the same time must be unequal. This is achieved as before.

- Given an element-nondeterministic as described in the definition of the syntax of ASMs and an expanded state $(\mathcal{A}, \xi)$. Let $M := \{\overline{a} \in \mathrm{domain}(\mathcal{A})^r : a_1, ..., a_r$ are non-reserve and $\mathcal{A} \models \alpha[\overline{z}/\overline{a}]\}$.

  $\Delta^{\mathrm{N}}(R, \mathcal{A}, \xi) = \bigcup_{\overline{a} \in M} \Delta^{\mathrm{N}}(R_0, \mathcal{A}, \xi[\overline{z} \to \overline{a}])$ 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 nondeterminism is called *element-nondeterminism*.

- Given a rule-nondeterministic as described in the definition of the syntax of ASMs and an expanded state $(\mathcal{A}, \xi)$. Then $\Delta^{\mathrm{N}}(R, \mathcal{A}, \xi) = \Delta^{\mathrm{N}}(R_1, \mathcal{A}, \xi) \cup \Delta^{\mathrm{N}}(R_2, \mathcal{A}, \xi)$.

Import isomorphism. The successor of a structure $\mathcal{A}$ with respect to an import rule whose inner rule is deterministic and not using import is not unique (there are infinitely many) but all successors are pairwise import isomorphic where import isomorphic means the following.

For a pair of states $\mathcal{B}, \mathcal{C}$ with $|\mathrm{ad}(\mathcal{B}) - \mathrm{ad}(\mathcal{C})| \leq 1$ and $|\mathrm{ad}(\mathcal{C}) - \mathrm{ad}(\mathcal{B})| \leq 1$, consider the mapping $h : \mathrm{domain}(\mathcal{B}) \to \mathrm{domain}(\mathcal{C})$, defined by

$$h(x) = \begin{cases} x & : \quad x \in \mathrm{ad}(\mathcal{A}) \vee x \notin \mathrm{ad}(\mathcal{B}) \\ y & : \quad \text{else} \end{cases} \quad \text{where } y \in \mathrm{ad}(\mathcal{C}) - \mathrm{ad}(\mathcal{B}).$$

If h is an isomorphism then $\mathcal{B}$ and $\mathcal{C}$ are import isomorphic. E.g., up to import isomorphism, a state has one successor with respect to a deterministic program.

This consideration can directly be transfered to the use of more than one import rule. In the case of nondeterministic programs, one can partition the set of successors into finitely many subsets such that for each two states from the subset there is an import isomorphism.

Run. Let ρ be a sequence $(\mathcal{A}_i)_{i<\lambda}$ of states where λ ranges over the natural numbers and the first infinite ordinal ω. ρ is a run of P on I if $\mathcal{A}_i \vdash_P \mathcal{A}_{i+1}$ for all i with $i + 1 < \lambda$ and $A_0 = I$.

Computation graph. The computation graph $C_\Pi(\mathcal{I})$ of an input structure $\mathcal{I}$ and an ASM-program Π is of the form $\mathcal{M} = (\mathcal{F}, D, I)$ where $\mathcal{F} = (W, R, s)$ is a transition system with source s, D is a non-empty set, the domain of $\mathcal{M}$, and I is a function associating with every world $w \in W$ a first-order structure and $I(s) = \mathcal{I}$,

$$I(w) = (D, P_0^{I(w)}, ..., f_0^{I(w)}, ...),$$

the state at w, in which $P_i^{I(w)}$, for each i, is a predicate on D of the same arity as P_i and $c_i^{I(w)}$ and $f_i^{I(w)}$, for each i, is a function on D of the same arity as f_i.

Ruv is true if, and only if, $I(u) \vdash_\Pi I(v)$. Furthermore, every label is only allowed to appear at most once in the computation graph (up to import-isomorphism).

In the case of a deterministic ASM, the computation graph collapses to a linear (discrete) order.

Query. An ASM query is a pair (Π, Q) consisting of an ASM program Π and a designated relation symbol Q. With every structure $\mathcal{A}$ on which the computation of Π eventually stagnates (i.e., no more changes occur in the states), the query (Π, Q) associates the result $(\Pi, Q)[\mathcal{A}]$, ie., the interpretation of Q as computed by Π on input $\mathcal{A}$.

For future use, we now define the atomic type of an element from the domain of a state.

The atomic type of an element x is the set of all locations satisfying the following properties:

1. if the location has the form $(R, \overline{y})$ where $R \in \Upsilon$ is a relation symbol and $\overline{y}$ is a tuple of elements then $\overline{y}$ contains x and the content of the location is true

2. if the location is a constant then its content is x

All other locations are not in the atomic type.

The following three definitions identify rather intuitive classes of ASMs by respectively disallowing the use of certain advanced constructs. We refer to these definitions later.

Definition 1. $\mathcal{D}$ denotes the class of all deterministic ASMs not using import.

Definition 2. $\mathcal{DI}$ denotes the class of all ASMs resulting from $\mathcal{D}$ by additionally allowing the use of import.

Definition 3. $\mathcal{ND}$ denotes the class of all ASMs resulting from $\mathcal{D}$ by additionally allowing the choice of a tuple of elements from the domain of a state.

Examples of Abstract State Machines

In this chapter, we consider two examples of an ASM. The first one is rather simple and has an introductory character. In the second example, we demonstrate how to simulate Turing machines by means of ASMs.

Modelling a Simple Algorithm

As a first example, we consider the ASM Π defined by the program

do-in-parallel
  forall $xy : Exy$ **do**
    $Txy :=$ true
  endforall
  forall $xyz : Exy$
    if Tyz **then** $Txz :=$ true **endif**
  endforall
enddo

The states of Π are undirected graphs with the edge relation E and an additional binary relation T. T is intended to be initially empty. When Π halts (and T has been initially empty), T corresponds to the transitive closure of the graph that has been the input.

Π constructs the transitive closure stepwisely. After the first execution step, T equals E. In the next step, we add all pairs of vertices with distance two. In the n-th step, all pairs of vertices with distance n are added to T (if they are not yet contained in T). After d steps, where d corresponds to the diameter of the input graph, no more changes can be observed in T. We say that Π halts.

Simulation of a Turing Machine

In this section, we give a simulation of a Turing machine by an ASM. We will use this simulation later in this work in order to prove some undecidability results

and refer to it as the basic simulation.

First, we recall the basic definition of Turing machine in order to use the denotation later.

Definition 4. 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 a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is an element $w \in \Sigma^*$. As we assume that this computation model is well known, there is not any further explication about this definition or its meaning.

A configuration can be described by the current control state, the current position of the head and the content of the tape.

Now, we give a kind of "basic simulation" of a Turing machine by an ASM. It can be adapted in the different contexts.
The states of the simulating ASM are structures of the vocabulary

$$\{\text{currentControl}, \text{Head}, \text{content}, \text{final}, \text{Succ}, \text{Pred}, \text{max}\} \cup Q \cup \Sigma$$

(furthermore it contains the obligatory function symbols as mentioned in the basic definitions) where the elements of Σ and of Q are constant symbols. The intended meanings of the functions are the following.

Succ: This is intended to be a successor function on the active domain.

Pred: This is intended to be the predecessor function corresponding to Succ.

max: This gives the maximal element of the active domain (with respect to the order indicated by Succ).

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

head: This nullary function provides the current position of the head of the simulated Turing machine (consequently, if the head is on position i then head is equal to the i-th successor of the first element of the active domain of the current state).

content: This unary function provides for every field of the tape of the simulated Turing machine its content. Therefore, content is a mapping from the active domain to Σ. 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) = undefined.

Intentionally, the boolean constant Mode is initialized by true and Output is only important in the case of a Boolean output.

The intended set of 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 $q_2 \notin F$, $x \in \{-1, 0\}$ define the rule

if Mode $\wedge$ currentControl $= q_1 \wedge$ content(Head) $= \sigma_1$ **then**
  do-in-parallel
    currentControl $:= q_2$
    content(Head) $:= \sigma_2$
    Head $:= \begin{cases} \text{Pred(Head)} & : \quad x = -1 \\ \text{Head} & : \quad x = 0 \end{cases}$
  enddo
endif

For every tuple $(q_1, \sigma_1, q_2, \sigma_2)$ such that $\delta(q_1, \sigma_1) = (q_2, \sigma_2, 1)$ and $q_2 \notin F$ define the following rule.

```
if Mode ∧ currentControl = q1 ∧ content(Head) = σ1 then
  do-in-parallel
    currentControl := q2
    content(Head) := σ2
    if Head = max then
      import v
        max := v
        Succ(max) := v
        Pred(v) := max
        Head := v
      endimport
    endif
  enddo
endif
```

For every tuple $(q_1, \sigma_1, q_2, \sigma_2)$ such that $\delta(q_1, \sigma_1) = (q_2, \sigma_2, x)$ and $q_2 \in F$, the simulation can be done analogously but nullary relation Mode has to be updated to false. This additional update indicates that the Turing-machine as well as the ASM halt.

The Turing machine is simulated by the parallel composition of all these rules.

Definition 5. For a Turing machine T, we denote by Π_T the simulating ASM as described above.

Part I

Guarded Abstract State Machines

1 Exploiting Concepts from Mathematical Logic

1.1 Classical Decidable Fragments of First-Order Logic

In the introduction of this thesis, we have already addressed the general verification problem and slicing of ASMs. Though they will be formally introduced later, we already recognize from their informal descriptions that both problems are strongly connected to decidability problems in mathematical logic.

In order to demonstrate this connection, we consider the ASM Π defined by the program **if** φ **then** res := true **endif** where φ is an arbitrary first-order formula not involving the nullary relation symbol res.

There is a computation of Π not satisfying the temporal formula (X res) (stating that res holds in the next state) if, and only if, $\neg\varphi$ is finitely satisfiable.

Similarly, computing a minimal slice for an ASM in $\mathcal{D}$ is directly connected to the satisfiability problem for first-order logic. **Skip** is a minimal slice of Π relative to the evaluation of res if, and only if, φ is not satisfiable. Furthermore, **Skip** is the only such minimal slice of Π in this case.
Consequently, the undecidability of the finite satisfiability problem for first-order logic FO implies the undecidability of the general verification problem and non-computability of a minimal slice for ASMs in $\mathcal{D}$.

A further witness for the connection to mathematical logic is that many proofs of (un)decidability resp. (non-)computability results are reductions of the considered problems to some (finite) satisfiability problem or vice versa.

The above observations suggest to use decidable instances of problems in mathematical logic to obtain solvable instances of decision resp. computation problems for ASMs by introducing appropriate restrictions for ASMs. A canonical candidate for such a problem is the (finite) satisfiability problem. In the remainder of this section, we review some of the most common decidable case of the (finite) satisfiability problem for first-order logic. They are obtained by rather straight restrictions on the use of quantifiers, variables, relations and functions.

The first such fragments that have been identified are the so-called prefix classes and fragments obtained by restrictions on the vocabulary.

Prefix classes are classes of first-order formulae in prenex normal form whose quantifier prefix satisfies some condition usually given by a regular expression. An example for such an expression is $\exists^*\forall^*$. The meaning of this prefix condition is that we allow only formulae of the form $\exists x_1 \ldots \exists x_m \forall y_1 \ldots \forall y_n \ \varphi$ where φ is a quantifier free formula.

In order to derive similar results for ASMs, we have to put appropriate restrictions on the ASM prefixes. The prefix of an ASM includes the quantifiers in the guards of the conditional rules, the quantifiers in the guards of the forall-rules and the use use of **forall** itself. Consequently, if we try to profit from the prefix classes of FO (resp. the decidability of the satisfiability problem) we must formulate a condition similar to the prefix conditions. These restrictions are often much stronger than the original ones.

Restrictions on the vocabulary are usually upper bounds on the maximal number and on the arities of the relation and function symbols. Examples are the forbiddance of function symbols or the use of relation symbols of arity $\geq n$ for some natural number n. The monadic fragment (i.e., the fragment where we allow only unary relation symbols in the vocabulary) is an example for a decidable instance of the satisfiability problem of FO obtained in this way. Later in this thesis, we will also consider the respective restriction on ASMs and it is clear that this restriction is rather strong. Nevertheless, it might suffice in some cases to consider such vocabularies.

A third kind of restrictions that we mention briefly are restrictions on the use of variables. An example is the two-variable fragment. This fragment contains exactly those FO-formulae that use at most two distinct variables. Precisely the same restriction leads to a class of ASMs with a rather low expressive power.

In the remainder of this part, we consider a further fragment FO that is a decidable instance of the (finite) satisfiability problem. The idea differs substantially from the above ones.

1.2 Guarded Fragments

In [3], an approach to decidable instances of the satisfiability problem for first-order logic has been presented that is essentially different from the ones mentioned in the preceding chapter. It originates from an investigation of the reason for the good model-theoretic properties of propositional modal logic.

Modal logics are used in many areas in computer science, e.g. for the verification of hardware and software systems, for knowledge representation, in databases and in artificial intelligence. The most important reason for the successful application of these logics is that they provide a good balance between expressive

power and computational complexity.

Informally, modal logic extends propositional logic by unary modal operators with which one can build new formulae $\langle a\rangle\psi$ resp. $[a]\psi$ from a formula ψ. a is any element from a set of actions A. The models of ML formulae are labeled transition systems with a distinguished node s. $\langle a\rangle\psi$ holds if there exists a successor of v such that ψ holds for this successor. $[a]\psi$ holds if ψ holds for all successors of v.

Propositional modal logic is formally defined as follows.

Definition 1.1 (Syntax of propositional modal logic). The set ML of modal-logic formulae (with actions from the set A and atomic properties P_i, $i \in I$) is inductively defined as follows.

- all propositional formulae with propositional variables P_i, $i \in I$, belong to ML.
- if $\varphi, \psi \in \text{ML}$ then $\neg\psi, (\psi \vee \varphi), (\psi \wedge \varphi) \in \text{ML}$
- if $\psi \in \text{ML}$ and $a \in A$ then $\langle a\rangle\psi, [a]\psi \in \text{ML}$

Models of formulae from ML are labeled transition systems of the form $\mathcal{T} = (V, (E_a)_{a\in A}, (P_i)_{i\in I})$ with domain V (the states of the transition system), binary relations $E_a \subseteq V \times V$ $(a \in A)$ (describing the transitions between the states) and unary relations $P_i \subseteq V$ $(i \in I)$ (giving properties of the states). If a transition system $\mathcal{T}$ together with some distinguished node $v \in V$ is a model of an ML-formula φ then we write $\mathcal{T}, v \models \varphi$.

The semantics of ML is defined by induction on the formulae. For such a transition system $\mathcal{T}$ and $v \in V$ $\mathcal{T}, v \models P_i$ if, and only if, $v \in P_i$. The semantics of the boolean connectives is as usual. Furthermore, $\mathcal{T}, v \models [a]\psi$ holds if, and only if, for all $w \in V$ with $(v, w) \in E_a$ $\mathcal{T}, w \models \psi$ holds and $\mathcal{T}, v \models \langle a\rangle\psi$ if, and only if, there exists $w \in V$ with $(v, w) \in E_a$ and $\mathcal{T}, w \models \psi$.

Though modal logic is an extension of propositional logic, it can be considered as a fragment of first-order logic, the modal fragment. The reason is that formulae from ML are basically propositions about structures (namely, labeled transition systems). With this change in the point of view, we can translate any ML-formulae ψ into a first-order formula ψ^*. E.g., $\langle a\rangle\psi$ is translated to $\exists y(E_a xy \wedge \psi^*(y))$ and $[a]\psi$ is translated to $\forall y(E_a xy \rightarrow \psi^*(y))$ In [3], H. Andréka, J. van Benthem and I. Németi have considered this translation a bit closer and detected that in the modal fragment, all quantifications are guarded by some atomic formula. This observation then leads to the definition of the guarded fragment of first-order logic.

Definition 1.2. Let Υ be a vocabulary containing only constant and relation symbols. The guarded fragment GF of first-order logic is defined inductively as follows:

1. Every relational atomic formula (possibly an equation) belongs to GF.

2. GF is closed under propositional connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.

3. If $\psi(\overline{x}, \overline{y})$ is a formula in GF then $\exists \overline{y}(\alpha(\overline{x}, \overline{y}) \wedge \psi(\overline{x}, \overline{y}))$ and $\forall \overline{y}(\alpha(\overline{x}, \overline{y}) \rightarrow \psi(\overline{x}, \overline{y}))$ belong to GF provided that $\text{free}(\psi) \subseteq \text{free}(\alpha) = \overline{x} \cup \overline{y}$ and $\alpha(\overline{x}, \overline{y})$ is a relational atomic formula.

Here, $\text{free}(\psi)$ means the set of all free variables of ψ.

Since the introduction of GF in [3], several guarded logics have been defined. They all have in common that every first-order quantifier is relativized by a guard. I.e., they have the form $\exists \overline{y}(\alpha(\overline{x}, \overline{y}) \wedge \psi(\overline{x}, \overline{y}))$ or $\forall \overline{y}(\alpha(\overline{x}, \overline{y}) \rightarrow \psi(\overline{x}, \overline{y}))$, $\text{free}(\psi) \cup \{\overline{x}\} \subseteq \text{free}(\alpha)$. In GF, the guards are atomic formulae.

As the translation of propositional modal logic to first-order logic uses only guarded quantification, the modal fragment is a subset of GF. The guarded fragment generalizes the modal fragment as there is no restriction to the number of variables or the arities of the relations. Based on a number of results (see [3], [13]), Andréka, van Benthem, and Németi assume that the guarded nature of quantification is the main reason for the good model-theoretic and algorithmic properties of modal logic. Generalizations of GF are the loosely guarded fragment LGF of FO (see [35]) and the clique-guarded fragment CGF of FO (see [15]). GF is strictly contained in CGF and every sentence in LGF is equivalent to a sentence in CGF.

As already mentioned, the clique guarded fragment of first-order logic is an expansion of GF.

In this definition, the notion of the Gaifman graph of a structure is used. The Gaifman graph of a structure $\mathcal{A}$ is an undirected graph which has the same domain as $\mathcal{A}$. There is an edge between two elements of the domain if, and only if, they coexist in some atomic fact in $\mathcal{A}$.

Definition 1.3. The Gaifman graph of a relational structure $\mathcal{A}$ (with universe A) is the undirected graph $G(\mathcal{A}) = (A, E^{\mathcal{A}})$ where

$$E^{\mathcal{A}} = \{(a, a') : \quad a \neq a', \text{ there exists a ground atomic formula } \mathcal{A} \models \alpha(b_1, ..., b_n) \text{ such that } a, a' \in \{b_1, ..., b_n\}\}$$

For each finite vocabulary Υ and each $k \in \omega$ (where ω is the first infinite ordinal), there is a positive, existential first-order formula $\text{clique}(x_1, ..., x_k)$ such that, for every Υ-structure $\mathcal{A}$ and all $a_1, ..., a_k$ in the universe of $\mathcal{A}$

$$\mathcal{A} \models \text{clique}(a_1, ..., a_k) \text{ if, and only if, } a_1, ..., a_k \text{ induce a clique in } G(\mathcal{A})$$

In the clique guarded fragment of first-order logic, CGF, all quantifiers are relativized by clique guards.

Definition 1.4. A clique guard is any formula clique($\overline{x}$) implying that $\overline{x}$ induces a clique in the Gaifman graph. I.e., if $\mathcal{A} \models \text{clique}(\overline{a})$ then $(a_i, a_j) \in E^{\mathcal{A}}$ for all $i \neq j$.

Definition 1.5. Let Υ be a vocabulary containing only constant and relation symbols. The clique-guarded fragment CGF of first-order logic is defined inductively as follows:

1. Every relational atomic formula (possibly an equality) belongs to CGF.
2. CGF is closed under propositional connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.
3. If $\psi(\overline{x}, \overline{y})$ is a formula in CGF then the formulae $\exists\overline{y}(\text{clique}(\overline{x}, \overline{y}) \wedge \psi(\overline{x}, \overline{y}))$ and $\forall\overline{y}(\text{clique}(\overline{x}, \overline{y}) \rightarrow \psi(\overline{x}, \overline{y}))$ belong to CGF provided that $\text{free}(\psi) \subseteq \text{free}(\text{clique}) = \overline{x} \cup \overline{y}$ and $\text{clique}(\overline{x}, \overline{y})$ is a clique guard.

Here, free(ψ) means the set of all free variables of ψ.

Definition 1.6. Let $\mathcal{E}$ be a logic extending FO and $\mathcal{L} \subseteq \mathcal{E}$ a fragment of $\mathcal{E}$.
GF($\mathcal{L}$) $\subseteq \mathcal{L}$ consists of all formulae in $\mathcal{L}$ in which every first-order quantifier is relativized by an atomic guard.

As already mentioned in the preceding chapter, concepts of restrictions on first-order logic can be transfered to ASMs. The definition of the guarded fragment of first-order logic motivates the following definition. Essentially, every forall-rule or choice-rule is relativized by an atomic formula.

Definition 1.7. The guarded fragment GF($\mathcal{C}$) of a class $\mathcal{C}$ of ASMs is the subset of $\mathcal{C}$ such that every element Π of GF($\mathcal{C}$) satisfies the following conditions:

- the vocabularies contain only relation and constant symbols
- the left-hand side and the right-hand side of any update rule are boolean-valued terms
- every guard of an if-clause in Π is in the guarded fragment of first-order logic
- every forall-rule has the form **forall** $\overline{x} : \alpha(\overline{x}, \overline{y})$ R **endforall** where $\alpha(\overline{x}, \overline{y})$ is an atomic formula and $\text{free}(R) \subseteq \text{free}(\alpha) = \{\overline{x}, \overline{y}\}$
- every choice-rule (choice of an element from the domain) has the form **choose** $\overline{x} : \alpha(\overline{x}, \overline{y})$ R **endchoose** where $\alpha(\overline{x}, \overline{y})$ is an atomic formula and $\text{free}(R) \subseteq \text{free}(\alpha) = \{\overline{x}, \overline{y}\}$

2 Computational Power of Guarded Abstract State Machines

Though we have already presented the definition of GF($\mathcal{C}$) for an arbitrary class of ASMs $\mathcal{C}$, we give the definition of GF($\mathcal{D}$) separately in order to clarify this special case. From here on, the main focus is on GF($\mathcal{D}$).

Definition 2.1. GF($\mathcal{D}$) is the class of all ASMs $\Pi \in \mathcal{D}$ such that every element Π of GF($\mathcal{D}$) uses only relation and constant symbols and satisfies the following conditions:

- the left-hand side and the right-hand side of any update rule are boolean-valued terms
- every guard of an if-clause in Π is in the guarded fragment of first-order logic
- every forall-rule has the form **forall** $\overline{x} : \alpha(\overline{x}, \overline{y})$ R **endforall** where $\alpha(\overline{x}, \overline{y})$ is an atomic formula and free$(R) \subseteq$ free$(\alpha) = \{\overline{x}, \overline{y}\}$

In order to achieve a basic understanding of this class, we consider the special case of graphs as states, i.e., the vocabulary contains only the binary relation symbol E.

Let $\Pi \in$ GF($\mathcal{D}$) with vocabulary $\{E/2\}$ and $\mathcal{A} = (V, E)$ be a state of Π. Then for all states $\mathcal{B} = (V, E')$ of Π with $\mathcal{A} \vdash_\Pi^* \mathcal{B}$, $E' \subseteq E$.

The reason is that we can only change the interpretation of E on pairs (x, y) that are connected via some edge. As soon as x and y are no more connected, we have no possibility to access this pair.

These considerations can be generalized to arbitrary ASMs from GF($\mathcal{D}$) over relational vocabulary with relations of arbitrary arity and tuples of arbitrary length (bounded by the maximal arity of the relation symbols in the vocabulary). In this case, the respective components of a tuple have to coincide in a tuple contained in some relation.

In the next section, we formalize the above properties and some further basic properties of GF($\mathcal{D}$).

2.1 Basic Properties of Guarded ASMs

In order to formulate some basic properties of guarded ASMs that increase the understanding of GF($\mathcal{D}$), we use the notion of the Gaifman graph of a structure (see definition 1.3).

The first lemma is a rather direct consequence of the restrictions in GF($\mathcal{D}$).

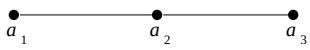
Lemma 2.2. Let $\Pi \in \mathrm{GF}(\mathcal{D})$ not using constants, $\mathcal{A}$ be a state of Π, and $G_\mathcal{A} = (V_\mathcal{A}, E^\mathcal{A})$ be its Gaifman graph. Furthermore, let $\mathcal{B}$ be the state of Π with $\mathcal{A} \vdash_\Pi \mathcal{B}$ and let $G_\mathcal{B} = (V_\mathcal{B}, E^\mathcal{B})$ be its Gaifman graph. Then $E^\mathcal{B} \subseteq E^\mathcal{A}$ holds.

Proof. Let $\Pi \in \mathrm{GF}(\mathcal{D})$ not using constants and $\mathcal{A}$ be a state of Π with Gaifman graph $G_\mathcal{A} = (V_\mathcal{A}, E^\mathcal{A})$. Let (a_1, a_2) be a pair of elements from the domain of $\mathcal{A}$ resp. from $V_\mathcal{A}$.

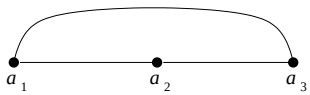
Assume that $G_\mathcal{A} \models \neg E a_1 a_2$. Then there exist no relation R from the vocabulary Υ of Π and no tuple $\bar{b}$ containing a_1 and a_2 such that $\mathcal{A} \models R\bar{b}$. In Π, elements from the domain of a state can only be addressed via variables and all variables have to be bounded because a program is not allowed to contain free variables. In GF($\mathcal{D}$), the only possibility to bind variables is the use of **forall** and every forall rule has to be relativized by an atomic formula. Therefore, the pair (a_1, a_2) can not be accessed by Π. Note that this does not exclude the separate access of both elements. Consequently, it is not possible that in the successor state $\mathcal{B}$ of $\mathcal{A}$, there exist a relation P and a tuple $\bar{c}$ containing a_1 and a_2 such that $\mathcal{B} \models P\bar{c}$. Therefore, $G_\mathcal{B} \models \neg E a_1 a_2$.

As a_1 and a_2 have been chosen arbitrarily, $E^\mathcal{B} \subseteq E^\mathcal{A}$. □

As a consequence of the above lemma, we obtain some non-computability results. E.g., the transitive closure is not computable by an ASM from GF($\mathcal{D}$). Consider for example undirected graphs. In this case, the Gaifman graph of a state $\mathcal{A}$ is received from $\mathcal{A}$ by removing all self-referring edges. The Gaifman graph $G_\mathcal{A} = (V_\mathcal{A}, E^\mathcal{A})$ of $\mathcal{A}$ given by



is essentially the state itself. The transitive closure of the above graph is



Again, the Gaifman graph $G_\mathcal{B} = (V_\mathcal{B}, E^\mathcal{B})$ is essentially the state $\mathcal{B}$ itself. But $E^\mathcal{B} \not\subseteq E^\mathcal{A}$ as $(a_1, a_3) \notin E^\mathcal{A}$ (but $(a_1, a_3) \in E^\mathcal{B}$).

The converse of lemma 2.2 is not true.

Lemma 2.3. There exists a relational ASM $\Pi \in \mathcal{D}$ that is not equivalent to an ASM from GF($\mathcal{D}$) and for all states $\mathcal{A}$ and $\mathcal{B}$ of Π with Gaifman graphs $G_{\mathcal{A}} = (V_{\mathcal{A}}, E^{\mathcal{A}})$ and $G_{\mathcal{B}} = (V_{\mathcal{B}}, E^{\mathcal{B}})$ the following holds. If $\mathcal{A} \vdash_{\Pi} \mathcal{B}$ then $E^{\mathcal{B}} \subseteq E^{\mathcal{A}}$.

Proof. In order to prove the lemma, we give a simple example of an ASM $\Pi \in \mathcal{D}$ over the vocabulary $\{E/2, P/0\}$ (E is a binary relation symbol and P is a nullary relation symbol).

$$\textbf{if } \forall x\ y\ z\ (Exy \wedge Eyz \rightarrow Exz) \textbf{ then } P := \text{false } \textbf{endif}$$

It is rather easy to see that for all states $\mathcal{A}$ and $\mathcal{B}$ with Gaifman graphs $G_{\mathcal{A}} = (V_{\mathcal{A}}, E^{\mathcal{A}})$ and $G_{\mathcal{B}} = (V_{\mathcal{B}}, E^{\mathcal{B}})$ the following holds. If $\mathcal{A} \vdash_{\Pi} \mathcal{B}$ then $E^{\mathcal{B}} \subseteq E^{\mathcal{A}}$. The reason is that we have only negative updates in Π.

Now, we prove that there is no ASM $\Pi' \in \text{GF}(\mathcal{D})$ that is equivalent to Π. The proof is done by contradiction. Assume that there exists an ASM $\Pi' \in \text{GF}(\mathcal{D})$ that is equivalent to Π.
We first demonstrate how we can rewrite Π' such that we obtain an equivalent ASM from GF($\mathcal{D}$) of the form

$$\textbf{if } \psi \textbf{ then } P := \text{false } \textbf{endif}$$

As Π' is equivalent to Π, Π' cannot change other relations than P and P is nullary. Consequently, if Π' contains an update rule with another left-hand side than P then this update rule is never executed. Therefore, we can remove this update rule from Π' resp. replace it by **Skip**.

Furthermore, every update rule of the form $P := t$ where t is an arbitrary boolean-valued term can be replaced by

do-in-parallel
  if t **then** $P :=$ true **endif**
  if $\neg t$ **then** $P :=$ false **endif**
enddo

Again, as Π' is equivalent to Π and Π updates P to true in no case, the first conditional rule in the above parallel composition can be removed. Therefore, we obtain

$$\textbf{if } \neg t \textbf{ then } P := \text{false } \textbf{endif}$$

instead.

As a first consequence of the above observations we can assume that every update rule appearing in Π' is equal to $P :=$ false.

Another consequence of these observations is that we can assume that no variable appears in an update rule of Π'. Hence, the successive application of the following equivalence preserving rewriting rules to Π' leads to an ASM from GF($\mathcal{D}$) that is equivalent to Π' and thus to Π.

original rule	**equivalent rule**

forall $\overline{x} : \alpha(\overline{x}, \overline{y})$ **do**
  if $\varphi(\overline{x})$ **then** $P :=$ false **endif**
endforall
	if $\exists \overline{x}(\alpha(\overline{x}, \overline{y}) \wedge \varphi(\overline{x}))$ **then**
	  $P :=$ false
	endif

if φ **then**
  if ψ **then** $\tilde{\Pi}$ **endif**
endif
	if $\varphi \wedge \psi$ **then** $\tilde{\Pi}$ **endif**

do-in-parallel
  if φ **then** $P :=$ false **endif**
  if ψ **then** $P :=$ false **endif**
enddo
	if $\varphi \vee \psi$ **then** $P :=$ false **endif**

If these rules are applied until no more of them is applicable then we obtain an ASM from GF($\mathcal{D}$) of the form

if ψ **then** $P :=$ false **endif**

As this ASM is equivalent to Π, ψ is equivalent to $\forall xyz\ (Exy \wedge Eyz \rightarrow Exz)$ and therefore, ψ expresses that E is transitive. As the above ASM is in GF($\mathcal{D}$), ψ is in GF. Consequently, transitivity is expressible in GF. This is a contradiction as it is a well-known fact that transitivity is not expressible in GF. □

Note that this proof is essentially a reduction to the respective property of GF and therefore, this is a further witness for the strong connection between ASMs and logic.

Though we have seen some limits of GF($\mathcal{D}$) in this section, many interesting properties are nevertheless computable in GF($\mathcal{D}$). In order to demonstrate this, we compare the expressive power of GF($\mathcal{D}$) to the expressive power of other well-known and expressive mechanisms.

2.2 Guarded ASMs are Stronger than Datalog LITE

The first mechanism to which we compare GF($\mathcal{D}$) is Datalog LITE (see [12]). Datalog LITE is a deductive query language with a linear time model checking

algorithm. It is a guarded version of stratified Datalog additionally allowing a limited form of universal quantification in rule bodies.

Despite linear time evaluation, Datalog LITE is highly expressive. In [12], it has been proven that Datalog LITE is equivalent in expressive power to alternation-free guarded fixed point logic, a natural fragment of the guarded fixed point logic. Well-known logical formalisms such as propositional multi-modal logic, computation tree logic (CTL), the alternation free μ-calculus, each correspond to well-defined and syntactically simple fragments of Datalog LITE.

In the next section, we prove that, for $\mathrm{GF}(\mathcal{D})$, we can skip the restriction of alternation-freedom necessary for the result on Datalog LITE and even obtain a strict result. I.e., $\mathrm{GF}(\mathcal{D})$ is stronger than guarded (first-order) fixed point logic. Though this proposition already implies that $\mathrm{GF}(\mathcal{D})$ is stronger than Datalog LITE, we prove it elementarily. We expect to obtain a better insight to the connection between Datalog LITE and $\mathrm{GF}(\mathcal{D})$ in this way as the proof is much straightforward.

Before comparing the expressive power of $\mathrm{GF}(\mathcal{D})$ to the one of Datalog LITE, we give a brief introduction to Datalog and Datalog LITE.

A Datalog rule is an expression of the form $H \leftarrow B_1, \ldots, B_r$ where H, the head of the rule, is an atomic formula $Ru_1 \ldots u_s$, and $B_1, \ldots, B_r$, the body of the rule, is a collection of literals (i.e., atomic formulae or negated atomic formulae) of the form $Sv_1 \ldots v_t$ or $\neg Sv_1 \ldots v_t$ where $u_1, \ldots, u_s, v_1, \ldots, v_t$ are variables. The relation R is called the head predicate of the rule.

A basic Datalog program Π is a finite collection of rules such that none of its head predicates occurs negated in the body of any rule. The predicates that appear only in the bodies of the rules are called extensional or input predicates. Given a relational structure $\mathcal{A}$ over the vocabulary of the input predicates, the program computes, via the usual fixed point semantics, an interpretation for the head predicates. A Datalog query is a pair (Π, Q) consisting of a Datalog program Π and a designated head predicate Q of Π. With every structure $\mathcal{A}$, the query (Π, Q) associates the result $(\Pi, Q)[\mathcal{A}]$, i.e., the interpretation of Q as computed by Π on input $\mathcal{A}$.

A stratified Datalog program is a sequence $\Pi = (\Pi_0, \ldots, \Pi_r)$ of basic Datalog programs which are called the strata of Π, such that each of the head predicates of Π is a head predicate in precisely one stratum Π_i and is used as an extensional predicate only in higher strata Π_j for $j > i$. In particular, this means that

1. if a head predicate of stratum Π_j occurs positively in the body of a rule of a stratum Π_i, then $j \leq i$, and

2. if a head predicate of stratum Π_j occurs negatively in the body of a rule of stratum Π_i, then $j < i$.

The semantics of a stratified program is defined stratum per stratum. The extensional predicates of a stratum Π_i are either extensional in the entire program Π or are head predicates of a lower stratum. Hence, once the lower strata are evaluated, we can compute the interpretation of the head predicates of Π_i as in the case of basic Datalog programs.

A Datalog rule is monadic if each of its literals has at most one free variable. Note that this does not necessarily imply that only unary predicates are used. We permit the appearance of literals $\neg S v_1 \dots v_k$ with $k > 1$ that may contain repeated occurrences of a single variable. A Datalog rule is guarded if its body contains a positive atom (the guard of the rule) in which all free variables of the rule occur.

A generalized literal G is an expression of the form $\forall y_1 \dots y_n(\alpha \rightarrow \beta)$ where α and β are atoms, and $\text{free}(\beta) \subseteq \text{free}(\alpha)$. The free variables $\text{free}(G)$ of G are given by $\text{free}(\alpha) - \{y_1, \dots, y_n\}$.

A Datalog LITE program is a stratified Datalog program whose rules are either monadic or guarded and which contains (unnegated) generalized literals. The notions of guardedness and stratification are extended to generalized literals.

The proposition of the following lemma essentially says that $\text{GF}(\mathcal{D})$ is *s*trictly more expressive than Datalog LITE. The notion of ASM query allows to compare ASMs and Datalog LITE directly. A ASM query (Π_{ASM}, Q) is equivalent to a Datalog LITE query $(\Pi_{\text{Datalog}}, Q)$ if, and only if, for every state $\mathcal{A}$ (over an appropriate vocabulary) and every tuple $\overline{a}$ of elements from the domain of $\mathcal{A}$

$$\overline{a} \in (\Pi_{\text{ASM}}, Q)[\mathcal{A}] \text{ if, and only if, } \overline{a} \in (\Pi_{\text{Datalog}}, Q)[\mathcal{A}]$$

Consequently, we consider equivalence with respect to the result of a computation.

Another possibility would be to compare programs stepwisely. This means, we consider each state in a computation of an ASM and compare it to the corresponding state in the computation of the Datalog LITE program. The ASM and the Datalog LITE program are equivalent in the case where the states are identical in each step of every computation. But as we compare $\text{GF}(\mathcal{D})$ to a logic fragment (namely, the guarded fragment of first-order fixed-point logic) in the next section where we do not have this notion of equivalence, we restrict to equivalence of queries in this case. This means that we consider equivalence with respect to the result of a computation.

Lemma 2.4. Every Datalog LITE query is equivalent to a $\text{GF}(\mathcal{D})$ query. Conversely, there exists a $\text{GF}(\mathcal{D})$ query for which there is no equivalent Datalog LITE query.

Proof. First, we show that $\text{GF}(\mathcal{D})$ is at least as strong as Datalog LITE. The idea of this part is to translate Datalog LITE programs to ASMs from $\text{GF}(\mathcal{D})$. We construct this translation stepwisely by starting with translating Datalog rules.

Here, we have to distinguish between the two types of rules, namely monadic and guarded ones. We then proceed with the translation of basic Datalog programs. Essentially, the translation of a basic Datalog program is the parallel composition of the translations of the contained rules. Finally, we construct from this components a translation for a Datalog LITE program. The stratification requires an additional construct.

At the end, we prove strictness via an example consisting of an ASM for which there exists no corresponding Datalog LITE program.

In the remainder, we formally prove the proposition of the lemma.

The first step in the translation is to map every Datalog LITE rule R to an ASM rule $a(R) \in \mathrm{GF}(\mathcal{D})$ where $a(R)$ is defined as follows.

If R is monadic then it has the form $Px \leftarrow B$ where B is a sequence of literals each with at most one variable. Let S_x^R be the set of all literals with free variable x and S_o^R the set of all literals in B with variables distinct from x. Then $a(R)$ is the rule

forall x : true **do**
  if $\bigwedge_{\beta \in S_x^R} \beta \wedge \bigwedge_{y \in \mathrm{free}(R)-\{x\}} \left(\exists y \bigwedge_{\beta \in S_y^R} \beta \right)$ **then** $Px :=$ true **endif**
endforall

Else R is a guarded rule. Let g be the guard of R with $\mathrm{free}(g) = \{\overline{x}\}$, α be its head and B its body. Then $a(R)$ is the rule

forall $\overline{x} : g$ **do**
  if $\bigwedge_{\beta \text{ appears in } B} \beta$ **then** $\alpha :=$ true **endif**
endforall

where $\overline{x}$ is a tuple of variables containing exactly the free variables of g.

This completes the translation of single rule. In the next step, we define the translation of a Datalog program from the translations of the contained rules.

A basic Datalog program D consisting of the rules $R_1, \ldots, R_n$ is translated to

do-in-parallel
  $a(R_1)$
  $\vdots$
  $a(R_n)$
enddo

Let $D = (D_0, \ldots, D_r)$ be a Datalog LITE program whose strata are basic Datalog programs. In addition to those relation and constant symbols from the vocabulary of D, the vocabulary of the resulting ASM contains r nullary boolean constant

symbols $\text{computed}_0, \ldots, \text{computed}_{r-1}$ marking whether the computation of the i-th stratum is completed. D is translated to the ASM rule

do-in-parallel
  if $\neg\text{computed}_0$ **then**
    do-in-parallel
      $a(D_0)$
      if φ_0 **then** $\text{computed}_0 := \text{true}$ **endif**
    enddo
  endif
  A_1
  $\vdots$
  A_r
enddo

where A_i, $i \in \{1, \ldots, r\}$ abbreviates

if $\text{computed}_{i-1} \wedge \neg\text{computed}_i$ **then**
  do-in-parallel
    $a(D_i)$
    if φ_i **then** $\text{computed}_i := \text{true}$ **endif**
  enddo
endif

and the formula $\varphi_i, i \in \{1, \ldots, r\}$, intuitively expresses that the computation of the i-th stratum is completed. Formally, the formula $\varphi_i, i \in \{1, \ldots, r\}$ is given by

$$\bigwedge_{R \in \text{mon}(D_i)} \forall x \left[\left(\bigwedge_{\beta \in S_x^R} \beta \wedge \bigwedge_{y \in \text{free}(R) - \{x\}} \left(\exists y \bigwedge_{\beta \in S_y^R} \beta \right) \right) \rightarrow Px \right] \wedge$$

$$\bigwedge_{\substack{\alpha \leftarrow B \in \text{guarded}(D_i) \text{ with guard } g \\ \text{free}(g) = \{x_1, \ldots, x_k\}}} \forall x_1 \ldots x_k \left[g \rightarrow \left(\left(\bigwedge_{\beta \text{ is in } B \text{ but not } g} \beta \right) \rightarrow \alpha \right) \right]$$

where $\text{mon}(D_i)$ is the set of monadic rules in D_i and $\text{guarded}(D_i)$ is the set of guarded rules in D_i.

In the rest of this proof, we show that $\text{GF}(\mathcal{D})$ is *strictly* more expressive than Datalog LITE, i.e., there exists a $\text{GF}(\mathcal{D})$ query for which there is no equivalent Datalog LITE query. In order to prove strictness, we use the following lemma that has been proved in [12].

Let Π be a Datalog LITE program with head predicates $T_1, \ldots, T_r$, and let $T_1^{\mathcal{A}}, \ldots, T_r^{\mathcal{A}}$ be the relations computed by Π on an input structure $\mathcal{A}$. Then the Gaifman graph of the expanded structure $(\mathcal{A}, T_1^{\mathcal{A}}, \ldots, T_r^{\mathcal{A}})$ coincides with the Gaifman graph of $\mathcal{A}$.

There exists an ASM $\Pi \in \mathrm{GF}(\mathcal{D})$ whose vocabulary does not contain constant symbols and a state $\mathcal{A}$ of Π such that the Gaifman graph of the state $\mathcal{B}$ with $\mathcal{A} \vdash_\Pi \mathcal{B}$ and the Gaifman graph of $\mathcal{A}$ do not coincide.

In order to witness this, consider for example the ASM $\Pi \in \mathrm{GF}(\mathcal{D})$ over the vocabulary $\{E/2\}$ defined by

forall $xy : Exy$ **do**
  $Exy :=$ false
endforall

The states of this ASM are directed graphs. The Gaifman graph of a directed graph is essentially the corresponding undirected graph (resulting by renaming the relation) where the self-referring edges are removed.

Consider a state $\mathcal{A}$ of Π that is a directed graph with at least one non-self-referring edge. Its Gaifman graph contains at least one edge. Executing the above program at least once for $\mathcal{A}$, leads to the graph over the same domain resp. set of nodes but with no edge. Therefore, the Gaifman graph of the successor state contains no edge. Consequently, Π changes the Gaifman graph. □

2.3 Guarded ASMs are Stronger than Guarded Fixed Point Logic

One of the most important extensions of ML is the μ-calculus. It extends ML by least and greatest fixed points. This motivates to extend GF also by least and greatest fixed points leading to guarded fixed point logic μGF.

μGF is the second mechanism to which we compare $\mathrm{GF}(\mathcal{D})$. It has first been considered in [17]. Further considerations can be found in [14] and [15].

Definition 2.5. Guarded fixed point logic μGF is obtained by adding to GF the following rules for constructing fixed point formulae:

Let W be a k-ary relation variable and $\overline{x} = x_1 \ldots x_k$ be a tuple of distinct variables. Further, let $\psi(W, \overline{x})$ be a guarded formula where W appears only positively and not in guards. Moreover, we require that all the free variables of $\psi(W, \overline{x})$ are contained in $\overline{x}$. For such a formula $\psi(W, \overline{x})$, we can construct the formulae

$$\mathrm{lfp}_{\overline{x},W}\psi(\overline{x})$$
$$\mathrm{gfp}_{\overline{x},W}\psi(\overline{x})$$

The semantics of the fixed point formulae is the following. Given a structure $\mathcal{A}$ and a valuation χ for the free second-order variables in ψ, other than W, the formula $\psi(W, \overline{x})$ defines an operator on k-ary relations $W \subseteq A^k$, namely

$$\psi^{\mathcal{A},\chi}(W) := \{\overline{a} \in A^k : \mathcal{A}, \chi \models \psi(W, \overline{a})\}$$

As W occurs only positively in ψ, this operator is inductive. Therefore, it has a least fixed point $\mathrm{lfp}(\psi^{\mathcal{A},\chi})$ and a greatest fixed point $\mathrm{gfp}(\psi^{\mathcal{A},\chi})$. The semantics of least fixed point formulae is defined by

$$\mathcal{A}, \chi \models \left[\mathrm{lfp}(\psi^{\mathcal{A},\chi})\right](\overline{a}) \text{ iff } \overline{a} \in \mathrm{lfp}(\psi^{\mathcal{A},\chi})$$

and

$$\mathcal{A}, \chi \models \left[\mathrm{gfp}(\psi^{\mathcal{A},\chi})\right(\overline{a}) \text{ iff } \overline{a} \in \mathrm{gfp}(\psi^{\mathcal{A},\chi})$$

W is the fixed point variable of $[\mathrm{gfp}\ W\overline{x}.\psi](\overline{x})$ resp. $[\mathrm{lfp}\ W\overline{x}.\psi](\overline{x})$.

A μGF-formula is called alternation-free if there is no alternation between greatest and least fixed points inside the formula.

In order to shorten the proof given in the remainder of this work, we give the following definition in advance.

Definition 2.6. Let $\Pi \in \mathcal{D}$ be an ASM and $\mathcal{A}$ be a state of Π.

1. Let $(\mathcal{A}_i)_{i\in\mathbb{N}} = \mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \ldots$ be the run of Π on $\mathcal{A} = \mathcal{A}_0$. Assume that there exists an $i \in \mathbb{N}$ such that $\mathcal{A}_i$ coincides with $\mathcal{A}_{i+1}$. Let n be the smallest such number. Then $\mathcal{A}^*_\Pi$ denotes the state $\mathcal{A}_n$.

2. Furthermore, let $\overline{a}$ be a tuple of elements from the domain of $\mathcal{A}$ and R be a relation symbol whose arity corresponds to the length of $\overline{a}$. Then we say that $\overline{a} \in (\Pi, R)[\mathcal{A}]$ holds if, and only if, $\mathcal{A}^*_\Pi \models R\overline{a}$.

Note that an index as referred to in the above definition does not exist in every run of every ASM in $\mathcal{D}$. But in the following considerations we compare the expressive power of $\mathrm{GF}(\mathcal{D})$ and μGF and because of the monotonicity of the operators defined by the formulae in the fixed point constructs, we can restrict our considerations to ASMs where we can find such an index for all runs.

The notion of Datalog LITE query respectively ASM query provides the possibility to compare Datalog LITE respectively ASMs with μGF formulae. Essentially, the vocabulary of the Datalog LITE program respectively the ASM Π is supposed to contain a distinguished relation H that is initially empty. Then the query (Π, H) and the formula φ are said to be equivalent in the case that for every state $\mathcal{A}$ and every tuple $\overline{a}$ of elements from the domain of $\mathcal{A}$

$$\overline{a} \in (\Pi, H)[\mathcal{A}] \text{ iff } \mathcal{A} \models \varphi(\overline{a})$$

holds. Consequently, when the computation halts, the relation H contains exactly those tuples that satisfy the formula φ.

In [12], the following proposition has been proven.

Lemma 2.7. Every alternation-free sentence in μGF is equivalent to a Datalog LITE query. Conversely, every Datalog LITE query is equivalent to an alternation-free formula in μGF.

Lemma 2.4 and lemma 2.7 imply the following corollary.

Corollary 2.8. Every alternation-free sentence in μGF is equivalent to a GF($\mathcal{D}$) query. Conversely, there exists a GF($\mathcal{D}$) query for which there is no equivalent alternation free formula in μGF.

But this lemma can even be strengthened. Namely, it does not only hold for the alternation-free fragment of μGF but for the complete guarded fixed point logic.

The intuitive reason why we can abandon the restriction of alternation-freedom is rather simple. A Datalog LITE program is arranged in strata which can only be processed in a top-down order. Once a stratum has been left, it is not possible to return to it again. This restriction is the main reason for the necessity of alternation-freedom. For GF($\mathcal{D}$), we do not have any restriction on the order of executing parts of an ASM and can abandon alternation-freedom.

Theorem 2.9. Every sentence in μGF is equivalent to a GF($\mathcal{D}$) query. Conversely, there exists a GF($\mathcal{D}$) query for which there is no equivalent formula in μGF.

Proof. First, we prove that every sentence in μGF is equivalent to a GF($\mathcal{D}$) query. This is done via constructing for an arbitrary μGF-sentence and equivalent GF($\mathcal{D}$)-query.

W.l.o.g., we can assume that different quantifications use distinct variables. We can establish this form via simple renamings. If we do not assume this, we would have to distinguish in the following syntactically equal subformulae that occur in different places.
Let ψ be a μGF sentence, and φ a subformula of ψ. If φ is not a sentence then let α_φ be the guard of the innermost quantifier dominating φ.

Obviously, free(φ) $\subseteq$ free(α_φ). In the case where φ is a sentence, we can set $\alpha_\varphi := \text{true}$.

We now associate with every subformula φ of ψ an ASM Π_φ and a relation H_φ where H_φ has the same arity as φ. Let $\tilde{\mathcal{A}}$ be the state that results from $\mathcal{A}$ by extending the vocabulary with the new relation symbols introduced in the construction and initializing these relations with the empty set (resp. by false if they are nullary). Then our construction ensures that

$$\overline{a} \in (\Pi_\varphi, H_\varphi)[\mathcal{A}] \text{ iff } \mathcal{A} \models \varphi(\overline{a})$$

for every structure resp. state $\mathcal{A}$ and every tuple $\overline{a}$ that is guarded by α_φ. In the case that φ is a sentence (in particular for $\varphi = \psi$), this means that (Π_φ, H_φ) evaluates to true on $\mathcal{A}$ if, and only if, $\mathcal{A} \models \varphi(\overline{a})$.

Furthermore, we associate with every subformula φ of ψ which is non-atomic or uses a fixed point variable a relation computed$_\varphi$ of the same arity as φ indicating for a tuple $\overline{a}$ of elements from the domain of the considered structure (with given valuations of the relation variables) whether the computation of the valuation of φ has already been completed (for the tuple $\overline{a}$). For atomic formulae φ not involving a fixed point variable, computed$_\varphi\overline{x}$ has to be replaced by true in the following.

In order to simplify the construction, we eliminate the use of gfp by using the duality for greatest and least fixed points:

$$\mathrm{gfp}_{\overline{x},W}\xi(\overline{x},W) \equiv \neg\mathrm{lfp}_{\overline{x},W}\neg\xi(\overline{x},\neg W)$$

where $\xi(\overline{x},\neg W)$ is obtained by replacing subformulae $W\overline{t}$ by $\neg W\overline{t}$ in $\xi(\overline{x},W)$.

Now, we define for every μGF-sentence φ an ASM $\Pi_\varphi \in \mathrm{GF}(\mathcal{D})$ that is equivalent to φ in the already defined sense. Π_φ is defined by induction on φ.

- If φ is an atomic formula not using a fixed point variable then Π_φ consists of the single rule

  forall $\overline{x}$: $\varphi(\overline{x})$ **do**
  $H_\varphi\overline{x} := \mathrm{true}$
  endforall

  As H_φ is initialized by the empty relation, H_φ contains exactly those tuples of elements satisfying φ after executing the above updates.

- If φ is an atomic formula using a fixed point variable then Π_φ consists of the single rule

  forall $\overline{x}$: $H_\varphi\overline{x}$ **do**
  $H_\varphi\overline{x} := \mathrm{true}$
  computed$_\varphi\overline{x} := \mathrm{true}$
  endforall

  In this case, the valuation of φ may change. But we know that all fixed point operators are inductive. Therefore, it is not possible that a tuple was in a relation interpreting a fixed point variable and at some point it is removed except that the fixed point formula with the involved fixed point variable has to be evaluated from new (starting with the empty relation).

- If $\varphi = \nu \wedge \vartheta$ then Π_φ is the rule

  do-in-parallel
  Π_ν
  Π_ϑ
  forall $\overline{x}, \overline{y}$: $\alpha_\varphi(\overline{x}, \overline{y})$ **do**
  if $\text{computed}_\nu \overline{x}' \wedge \text{computed}_\vartheta \overline{x}''$ **then**
  if $H_\nu \overline{x}' \wedge H_\vartheta \overline{x}''$ **then** $H_\varphi \overline{x} :=$ true **endif**
  $\text{computed}_\varphi \overline{x} :=$ true
  endif
  endforall
  enddo

  ($\overline{x}'$ and $\overline{x}''$ are subtuples of $\overline{x}$)

  In order to compute the value of a conjunction of two formulae we require that the valuations of both formulae are already known resp. computed. The interpretation of the assigned relation $H_{\nu \wedge \vartheta}$ is the disjunction of the relations H_ν and H_ϑ. This disjunction is reflected by a conjunction.

- If $\varphi = \nu \vee \vartheta$ then Π_φ is the rule

  do-in-parallel
  Π_ν
  Π_ϑ
  forall $\overline{x}, \overline{y}$: $\alpha_\varphi(\overline{x}, \overline{y})$ **do**
  if $\text{computed}_\nu \overline{x}' \wedge \text{computed}_\vartheta \overline{x}''$ **then**
  if $H_\nu \overline{x}' \vee H_\vartheta \overline{x}''$ **then** $H_\varphi \overline{x} :=$ true **endif**
  $\text{computed}_\varphi \overline{x} :=$ true
  endif
  endforall
  enddo

  ($\overline{x}'$ and $\overline{x}''$ are subtuples of $\overline{x}$)

  The respective argumentation are completely analogous to the one for conjunction.

- If $\varphi = \neg\vartheta$ then Π_φ is the rule

  do-in-parallel
  Π_ϑ
  forall $\overline{x}, \overline{y}$: $\alpha_\varphi(\overline{x}, \overline{y})$ **do**
  if $\text{computed}_\vartheta \overline{x}$ **then**

if $\neg H_\vartheta \overline{x}'$ **then** $H_\varphi \overline{x} :=$ true
else $H_\varphi \overline{x} :=$ false **endif**
computed$_\varphi \overline{x} :=$ true
endif
endforall
enddo

($\overline{x}'$ is a subtuple of $\overline{x}$)

In this case, H_ϑ is simply complemented on the tuples of interest. The set of tuples we are interested in is defined by α_ϑ. In order to complement H_ϑ, this relation has to be computed on this set of tuples.

- If $\varphi = \forall \overline{y}(\alpha_\vartheta \to \vartheta)$ then Π_φ is the rule

do-in-parallel
forall $\overline{x}, \overline{y}$: $\alpha_\varphi(\overline{x}, \overline{y})$ **do**
Π_ϑ
if $\forall \overline{z}(\alpha_\vartheta(\overline{x}', \overline{z}) \to \text{computed}_\vartheta(\overline{x}'', \overline{z}'))$ **then**
if $\forall \overline{z}(\alpha_\vartheta(\overline{x}', \overline{z}) \to H_\vartheta(\overline{x}'', \overline{z}'))$ **then** $H_\varphi \overline{x} :=$ true **endif**
computed$_\varphi \overline{x}' :=$ true
endif
endforall
enddo

($\overline{x}'$ is a subtuple of $\overline{x}$, $\overline{x}''$ is a subtuple of $\overline{x}'$ and $\overline{z}'$ is a subtuple of $\overline{z}$)

When the value of the formula φ is determined, the valuation of ϑ has to be known for all important variable assignments. These are indicated by α_ϑ.

Note that the guards α_ϑ and α_φ are atomic formulae not using a fixed point variable. Therefore, they are static and we can use them directly in the ASM. It is not necessary to take H_{α_ϑ} or H_{α_φ}.

- If $\varphi = \exists \overline{y}(\alpha_\vartheta \wedge \vartheta)$ then Π_φ is the rule

do-in-parallel
forall $\overline{x}, \overline{y}$: $\alpha_\varphi(\overline{x}, \overline{y})$ **do**
Π_ϑ
if $\forall \overline{z}(\alpha_\vartheta(\overline{x}', \overline{z}) \to \text{computed}_\vartheta(\overline{x}'', \overline{z}'))$ **then**
if $\exists \overline{z}(\alpha_\vartheta(\overline{x}', \overline{z}) \wedge H_\vartheta(\overline{x}'', \overline{z}'))$ **then** $H_\varphi \overline{x} :=$ true **endif**
computed$_\varphi \overline{x}' :=$ true

endif
endforall
enddo

($\overline{x}'$ is a subtuple of $\overline{x}$, $\overline{x}''$ is a subtuple of $\overline{x}'$ and $\overline{z}'$ is a subtuple of $\overline{z}$)

The construction and argumentation are analogous to those for universal quantification.

- If $\varphi = [\mathrm{lfp}_{W,\overline{x}}\vartheta(W,\overline{x})](\overline{x})$ then, by the induction hypothesis, there is an ASM from $\mathrm{GF}(\mathcal{D})$ computing the update operator defined by $\vartheta(W,\overline{x})$. Let $\xi_1,\ldots,\xi_m$ be those subformulae of φ involving the fixed point variable W which do not have a fixed point constructor outermost. Furthermore, let $\rho_1,\ldots,\rho_n$ be the other subformulae of φ involving W. I.e., those subformulae which contain W and have a fixed point constructor outermost. Let Π'_φ be the following rule.

do-in-parallel
Π_ϑ
forall $\overline{x}$: $\alpha_\varphi(\overline{x},\overline{y})$ **do**

if computed$_\vartheta\overline{x}$ **then**

// the current computation of ϑ is completed. I.e., the
// computation of the valuation with the current values

if $\neg(W\overline{x} \leftrightarrow \vartheta(\overline{x}))$ **then**

// fixed point not yet reached

do-in-parallel
if ϑ **then** $W\overline{x} :=$ true **endif**

// update of interpretation of fixed point variable

// Compute the valuation of those formulae which use
// the fixed point variable W. The valuation of
// these formulae might have changed as W has
// changed. If such a formula has a fixed point
// constructor outermost then the respective fixed
// point has to be computed new. Until this compu-
// tation is not finished (i.e., the fixed point
// is not yet reached) the value of the respective

```
                // computed relation is false for all important tuples.

                forall ū_{ξ1} : H_{ξ1}(ū_{ξ1}) do
                   H_{ξ1}(ū_{ξ1}) := false
                endforall
                // ξ_i needs to be computed again with new
                // interpretation of W
                forall z̄_{ξ1} : α_{ξ1}(x̄^{ξ1}, z̄_{ξ1}) do
                   computed_{ξ1}(x̄^{ξ1'}, z̄'_{ξ1}) := false
                endforall
                ⋮
                forall ū_{ξm} : H_{ξm}(ū_{ξm}) do
                   H_{ξm}(ū_{ξm}) := false
                endforall
                forall z̄_{ξm} : α_{ξm}(x̄^{ξm}, z̄_{ξm}) do
                   computed_{ξm}(x̄^{ξm'}, z̄'_{ξm}) := false
                endforall

                forall z̄_{ρ1} : α_{ρ1}(x̄^{ρ1}, z̄_{ρ1}) do
                   computed_{ρ1}(x̄^{ρ1'}, z̄'_{ρ1}) := false
                endforall
                ⋮
                forall z̄_{ρn} : α_{ρn}(x̄^{ρn}, z̄_{ρn}) do
                   computed_{ρn}(x̄^{ρn'}, z̄'_{ρn}) := false
                endforall
             enddo
          endif
       endif
    endforall
    if ∀x̄(α_φ(x̄, ȳ) → (W(x̄) ↔ ϑ(x̄))) then
       forall x̄ : α_φ(x̄, y) do computed_φ(x̄) := true endforall
    endif
enddo
```

where $\overline{x}^{\xi_i}$ and $\overline{x}^{\rho_j}$ ($i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$) are subtuples of $\overline{x}$, $\overline{x}^{\xi_i'}$ is a subtuple of $\overline{x}^{\xi_i}$ ($i \in \{1, \ldots, m\}$), and $\overline{x}^{\rho_j'}$ is a subtuple of $\overline{x}^{\rho_j}$ ($j \in \{1, \ldots, n\}$). Furthermore $\overline{z}'_{\xi_i}$ is a subtuple of $\overline{z}_{\xi_i}$ ($i \in \{1, \ldots, m\}$) and $\overline{z}'_{\rho_j}$ is a subtuple of $\overline{z}_{\rho_j}$ ($j \in \{1, \ldots, n\}$).

Π_φ results from Π'_φ by replacing all occurrences of W and H_ϑ by H_φ.

The ASM Π_φ simulates the computation of the least fixed point stepwisely. The comments given in the ASM explain its work.

After the construction of Π_φ, we have to remove the updates of computed_α for all atomic formulae α not using a fixed point variable resp. we have to replace it by **Skip**. Remember that for atomic formulae φ not using a fixed point variable, $\text{computed}_\varphi \overline{x}$ has to be replaced by true.
Initial states are those which satisfy the three formulae

$$\bigwedge_{\xi \text{ is a subformula of } \varphi} \forall \overline{x}_\xi (\neg H_\xi \overline{x}_\xi),$$

$$\bigwedge_{\xi \text{ is a subformula of } \varphi} \forall \overline{x}_\xi (\neg \text{computed}_\xi \overline{x}_\xi),$$

$$\bigwedge_{\substack{V \text{ is a relation variable} \\ \text{appearing in } \varphi}} \forall \overline{y}_V (\neg V \overline{y}_V).$$

This completes the construction of the GF($\mathcal{D}$)-query equivalent to φ.

We still have to prove that there exists a GF($\mathcal{D}$) query for which there is no equivalent formula in μGF. But this is rather clear. There are mainly two constructs in GF($\mathcal{D}$) that can not be simulated in μGF. Each of the constructs alone would already imply the above proposition and therefore, we have at least two possibilities to prove it. We present them both in the remainder of this proof. Each of them exploits one of the additional constructs.

First, GF($\mathcal{D}$) admits positive and negative updates. This means that it is allowed to use updates of the form $\alpha :=$ false and updates of the from $\alpha :=$ true both in ASMs from GF($\mathcal{D}$). Therefore, the evaluation of the dynamic relations is not always inductive. It even does not need to stagnate. In contrast to this, the evaluation of a fixed-point variable is always inductive in μGF. As a consequence, such evaluations of dynamic relations can not be simulated in μGF.
A concrete example for such an ASM is the ASM defined by the program

do-in-parallel
  forall $\overline{x} : R\overline{x}$ **do**
    if $\neg Q\overline{x}$ **then** $Q\overline{x} :=$ true **endif**
  endforall
  forall $\overline{x} : R\overline{x}$ **do**
    if $Q\overline{x}$ **then** $Q\overline{x} :=$ false **endif**
  endforall
enddo

For every tuple $\overline{a}$ in the relation R, the valuation of Q changes in every steps from true to false or vice versa.

Second, the guards in μGF are not allowed to contain fixed-point variables. There

is an extension of μGF (we denote it by μGF^e in the following) that admits to use fixed-point variables in the guards. M. Otto has proven that there is a μGF^e-sentence that is not equivalent to a μGF-sentence. But in GF($\mathcal{D}$), dynamic relations are allowed to appear in the guards and with the translation of μGF-formula given in this proof, we have also shown that every sentence in μGF^e is equivalent to a GF($\mathcal{D}$)-query. This implies immediately the above proposition. □

Part II

Verification

3 Verifying and Specifiying Properties of Abstract State Machines

3.1 The Verification Problem

In this part, we investigate the decidability of the verification problem for ASMs. As we have already mentioned in the introduction to this thesis, there are basically two versions of the verification problem, namely the general and the restricted verification problem. Here, the focus is on the general verification problem.

The general verification problem deals with the question whether *all* computations of an ASM satisfy a property whereas the restricted verification problem corresponds to the question whether the computations of an ASM for a *fixed* input satisfy a property. This is formalized in the following definition.

Definition 3.1. Let $\mathcal{A}$ be a class of ASMs and $\mathcal{L}$ be a class of specification formulae. Then

- $\text{GVerify}(\mathcal{A}, \mathcal{L}) = \{(\Pi, \varphi) | \Pi \in \mathcal{A}, \varphi \in \mathcal{L}, C_\Pi(\mathcal{I}) \models \varphi \text{ for all input structures } \mathcal{I} \text{ of } \Pi\}$
  denotes the general verification problem for $\mathcal{A}$ and $\mathcal{L}$
- $\text{RVerify}(\mathcal{A}, \mathcal{L}) = \{(\Pi, \mathcal{I}, \varphi) | \Pi \in \mathcal{A}, \varphi \in \mathcal{L}, C_\Pi(\mathcal{I}) \models \varphi\}$ denotes the restricted verification problem for $\mathcal{A}$ and $\mathcal{L}$

The outline of this part is as follows. First, we give some examples of logics for specifying properties of ASMs and that we can substitute for the logic $\mathcal{L}$ in the preceding definition. Then we consider some simple restrictions on ASMs and these logics. Simple restrictions are meant in the sense that we forbid certain advanced construction elements as **forall**, **choose** or temporal operators in the logic. Though these decidable instances of the verification problems are rather weak, their expressive power may suffice in a number of cases. But because of the weakness of these instances, we have to consider other kinds of restrictions. In [30] and [32], M. Spielmann has already analyzed some of them. We briefly resume these results in chapter 5. In the remainder of this part, we present new decidable instances of the general verification problem. The first class of ASMs that we consider in this context is a well-defined fragment of $\text{GF}(\mathcal{D})$. The second

class is the class of monadic ASMs, i.e., the class of all deterministic ASMs not using **import** and whose vocabulary contains only unary relation symbols.

3.2 Specifying Properties of Abstract State Machines

In this section, we present some basic formalisms for specifying properties of ASMs. All presented formalisms are extensions of first-order logic by temporal constructs. They are divided into two classes of temporal logics, namely, linear time logics and branching time logics.

3.2.1 Linear Temporal Logic

Definition 3.2. Linear temporal first-order logic TLFO is obtained by adding to FO the following rules for constructing temporal formulae:

If φ and ψ are TLFO-formulae then φ U ψ and φ S ψ are formulae in TLFO.

Semantics. TLFO is interpreted in first-order temporal models of the form $\mathcal{M} = (\mathcal{F}, D, I)$ where $\mathcal{F} = (W, <)$ is a strict linear order representing the flow of time, D is a non-empty set, the domain of $\mathcal{M}$, and I is a function associating with every moment of time $w \in W$ a first-order structure

$$I(w) = (D, P_0^{I(w)}, ..., c_0^{I(w)}, ...),$$

the state at moment w, in which $P_i^{I(w)}$, for each i, is a predicate on D of the same arity as P_i and $c_i^{I(w)}$ is an element of D. It is required that $c_i^{I(w)} = c_i^{I(v)}$ for any $w, v \in W$. In the following, the superscript I is omitted (and therefore $P_i^w, c_i^w, \ldots$ will be written) if I is clear from the context.

An assignment in D is a function a from the set of variables to D. The truth-relation $(\mathcal{M}, w) \models^a \varphi$ (or simply $w \models^a \varphi$, if $\mathcal{M}$ is understood) in the model $\mathcal{M}$ under the assignment a is defined as follows (which corresponds to the usual definition):

- $w \models^a P_i(y_1, \ldots, y_l)$ if, and only if, $P_i^w(a(y_1), \ldots, a(y_l))$ is true in $I(w)$
- $w \models^a \varphi \wedge \psi$ if, and only if, $w \models^a \varphi$ and $w \models^a \psi$
- $w \models^a \neg\psi$ if, and only if, not $w \models^a \psi$
- $w \models^a \forall x\psi$ if, and only if, $w \models \psi$ for every assignment B in D that may differ from a only on x.

- $w \models^a \varphi \text{ S } \psi$ if, and only if, there is $v < w$ such that $v \models^a \psi$ and $u \models^a \varphi$ for every u in the interval $(v, w) = \{u \in W : v < u < w\}$.

  Informally, $w \models^a \varphi \text{ S } \psi$ states that there exists a state in the past where ψ held and since then φ has been true.

- $w \models^a \varphi \text{ U } \psi$ if, and only if, there is $v > w$ such that $v \models^a \psi$ and $u \models^a \varphi$ for every $u \in (w, v)$.

  Informally, $w \models^a \varphi \text{ U } \psi$ states that there exists a state in the future where ψ holds and until that state is reached φ holds.

Here, we are interested in discrete time and consequently, the focus is on the order $(W, <) = (\mathbb{N}, <)$. For continuous time see e.g. [4].

Abbreviations. We use the following abbreviations.

$$\begin{array}{lcll} \text{F}\varphi & = & \top \text{ U } \varphi & \text{(some time in the future)} \\ \text{G}\varphi & = & \varphi \wedge \neg\text{F}\neg\varphi & \text{(always)} \\ \text{X}\varphi & = & \bot \text{ U } \varphi & \text{(at the next moment)} \\ \text{P}\varphi & = & \bot \text{ S } \varphi & \text{(at the previous moment)} \end{array}$$

This logic is well suited to describe properties of deterministic ASMs.

3.2.2 Branching Time Logic

In contrast to the models in linear time logic, the states in a model of a formula from a branching time logic are allowed to have more than one successor. These logics are well suited to describe properties of nondeterministic ASMs.

Hennessy-Milner First-Order Logic

Hennessy-Milner first-oder logic results from first-order logic by adding the following rule to the construction rules for first-order logic:

> If φ is a Hennessy-Milner first-order formula then $\diamond\varphi$ is a Hennessy-Milner first-order formula.

We denote this set of formulae by HM .

Formulae from HM are interpreted over transition systems whose states are labeled by first-order structures. Examples for such transition systems are computation graphs of ASMs.

Formally, these structures have the form $\mathcal{M} = (\mathcal{F}, D, I)$ where $\mathcal{F} = (W, R, s)$ is a transition system with source s, D is a non-empty set, the domain of $\mathcal{M}$, and I is a function associating with every world $w \in W$ a first-order structure and $I(s) = \mathcal{I}$,

$$I(w) = (D, P_0^{I(w)}, \ldots, f_0^{I(w)}, \ldots),$$

the state at w, in which $P_i^{I(w)}$, for each i, is a predicate on D of the same arity as P_i and $c_i^{I(w)}$ and $f_i^{I(w)}$, for each i, is a function on D of the same arity as f_i.

Only for $\diamond$, the semantics has to be defined. The rest is defined as in first-order logic where the quantifiers are relativized to the active domain of a structure.

Given a structure $\mathcal{M} = (\mathcal{F}, D, I)$ as above where $\mathcal{F} = (W, R, s)$. Then $\mathcal{M} \models \diamond\varphi$ if, and only if, there exists a state $t \in W$ such that Rst and $(\mathcal{F}', D, I) \models \varphi$ where $\mathcal{F}' = (W, R, t)$.

Informally, $\diamond\varphi$ states that there exists a successor of the current state satisfying φ.

$\Box\varphi$ abbreviates $\neg \diamond \neg\varphi$. Informally, $\Box\varphi$ states that all successors of the current state satisfy φ.

Definition 3.3. The (temporal) nesting depth $\mathrm{nd}(\varphi)$ of a HM-formula is defined by induction on φ:

- if φ does not contain $\diamond$ then $\mathrm{nd}(\varphi) = 0$.
- $\mathrm{nd}(\neg\psi) = \mathrm{nd}(\psi)$
- $\mathrm{nd}(\exists x \psi(x)) = \mathrm{nd}(\psi(x))$
- $\mathrm{nd}(\psi_1 \vee \psi_2) = \max(\mathrm{nd}(\psi_1), \mathrm{nd}(\psi_2))$
- $\mathrm{nd}(\diamond\psi) = \mathrm{nd}(\psi) + 1$

Essentially the temporal nesting depth of a HM-formula corresponds to the maximal number of nested $\diamond$'s.

First-Order CTL

First-order CTL is an extension of Hennessy-Milner first-order logic by adding the following construct:

If φ_1 and φ_2 are first-order CTL formulae then $\mathrm{E}[\varphi_1 \mathrm{U} \varphi_2]$ is a first-order CTL formula.

We denote this set of formulae by CTL-FO.

Analogously to formulae from HM, formulae from CTL-FO are interpreted over labeled transition systems whose labels are first-order structures. Only for $\mathrm{E}[\varphi_1 \mathrm{U} \varphi_2]$, the semantics has to be defined. The rest is defined as in Hennessy-Milner first-order logic.

Given such a transition system $\mathcal{M} = (\mathcal{F}, D, I)$ where $\mathcal{F} = (W, R, s)$. Then $\mathcal{M} \models \mathrm{E}[\varphi_1 \mathrm{U} \varphi_2]$ if there exists a sequence of worlds $t_1 \ldots t_n$, $t_i \in W$ for $i \in$

$\{0, \ldots, n\}$, such that $s = t_0$, Rt_it_{i+1} for all $i \in \{0, \ldots, n-1\}$, $(\mathcal{F}_i, D, I) \models \varphi_1$ where $\mathcal{F}_i = (W, R, t_i)$ for all $i \in \{0, \ldots, n-1\}$ and $(\mathcal{F}_n, D, I) \models \varphi_2$ where $\mathcal{F}_n = (W, R, t_n)$.

Informally, $\mathcal{M} \models \mathrm{E}[\varphi_1 \mathrm{U} \varphi_2]$ states that there exist a path and a state later on this path where φ_1 holds and until that state is reached φ_2 holds.

In the remainder, $\mathrm{EF}\varphi$ abbreviates $\mathrm{E}[\mathrm{true}\mathrm{U}\varphi]$.

4 Simple Restrictions

4.1 Simple Classes of Abstract State Machines

One idea for identifying verifiable classes of ASMs and formalisms is to forbid certain advanced construction elements or restricting their use in a rather simple way. Advanced construction elements are those that allow rules of the form **forall** $\overline{x} : \varphi(\overline{x})$ **do** Π **endforall**, the choice of an element from the domain of a state or **import**. By restricting ASMs in this way, we obtain the following classes of ASMs.

1. $\mathcal{C}_1$ is the class of all ASMs.

2. $\mathcal{C}_2$ is the class of all ASMs whose vocabulary contains only relation and constant symbols. In the programs, we allow only the use of update-rules, if-clauses, import and element-nondeterminism. (The use of forall is forbidden.)

3. $\mathcal{C}_3$ is the class of all ASMs whose vocabulary contains only relation and constant symbols. In the programs, we allow only the use of update-rules, if-clauses, forall-parallelism, import and rule-nondeterminism. (The only forbidden construct is element-nondeterminism.)

4. $\mathcal{C}_4$ is the class containing all deterministic ASMs using only update-rules, if-clauses and import.

5. $\mathcal{C}_5$ is the class of ASMs whose vocabulary contains only relation and constant symbols. The use of import and the use of rule-nondeterminism are allowed. The use of element-nondeterminism or forall is forbidden.

In this chapter, we investigate the decidability respectively the undecidability of the general and the restricted verification problem for these classes.

4.2 Decidability Results

4.2.1 Hennessy-Milner First-Order Logic

Though HM seems to be a very weak logic, there are cases where this logic is strong enough to express the intended property. This applies, for example, if we are only interested in a limited number of steps (e.g. whether a property is not satisfied before x steps are executed). Consequently, it makes sense to consider the question for which classes of ASMs the general respectively the restricted verification problem is decidable. Strong restrictions on the specification formalism allow weaker restrictions on the ASMs. In fact, for HM the restricted verification problem is decidable on all ASMs.

Theorem 4.1. RVerify$(\mathcal{C}_1, \mathrm{HM})$ is decidable.

Proof. In order to decide for a triple $(\Pi, \mathcal{I}, \varphi)$ with $\Pi \in \mathcal{C}_1$, $\varphi \in \mathrm{HM}$ and $\mathcal{I}$ an initial state of Π whether it is in RVerify$(\mathrm{HM}, \mathcal{C}_1)$, consider $\mathcal{I}$ and all states reachable from $\mathcal{I}$ within $\mathrm{nd}(\varphi)$ computation steps of Π (up to import isomorphism). The number of these states (up to import isomorphism) is finite because the active domain of $\mathcal{I}$ is finite and the program is finite. As the model checking problem for first-order logic (corresponding to the question whether a an FO-formula is satisfied in a structure that is part of the input) is decidable, RVerify$(\mathrm{HM}, \mathcal{C}_1)$ is decidable. □

Apart from decidability, we are interested in the complexity of RVerify$(\mathcal{C}_1, \mathrm{HM})$. For a state $\mathcal{I}$, an element-nondeterministic rule with guard g (containing r free variables) leads to at most as many successors as there are r-tuples of elements from $\mathrm{ad}(\mathcal{I})$ satisfying g. Let Π contain c such rules and i import rules. There are $\leq |\mathrm{ad}(\mathcal{A})|^{c \cdot r_{\max}}$ ($r_{\max}$ is the maximum of all numbers of free variables in the guards) successors of $\mathcal{I}$ (up to import-isomorphism). A state reachable from $\mathcal{I}$ within n steps has at most $n \cdot i + |\mathrm{ad}(\mathcal{I})|$ elements in its active domain. Therefore, within n steps, $\leq n \cdot (|\mathrm{ad}(\mathcal{I})| + n \cdot i)^{n \cdot c \cdot r_{\max}+1}$ states are reachable from $\mathcal{I}$ (up to import isomorphism).

As model checking for first-order logic can be done within exponential time, RVerify$(\mathrm{HM}, \mathcal{C}_1)$ is in TIME$(2^{O(p(\mathrm{nd}(\varphi)) \cdot (|\mathrm{ad}(\mathcal{I})| + n \cdot i)^{n \cdot c \cdot r_{\max}+1})})$ for a polynomial p.

Another interesting issue is the decidability of the corresponding verification problem. Because the finite satisfiability problem is undecidable for first-order logic, GVerify$(\mathrm{HM}, \mathcal{C}_1)$ is undecidable. We can prove this undecidability result via a reduction of GVerify$(\mathcal{C}_1, \mathrm{HM})$ to the finite satisfiability problem for first-order logic. A formula $\varphi \in \mathrm{FO}$ is satisfiable if, and only if, (**if** $\neg\varphi$ **then** Mode := true **endif**, □Mode) $\notin$ GVerify$(\mathcal{C}_1, \mathrm{HM})$.

In Part I, we have introduced restrictions for ASMs that are similar to GF. If we apply the restrictions to $\mathcal{C}_1$ and consider the guarded fragment of HM then obtain a decidability result.

Theorem 4.2. GVerify(GF($\mathcal{C}_1$), GF(HM)) is decidable.

Proof. We prove this proposition via a reduction to the finite satisfiability problem for the guarded fragment of first-order logic (GF). We construct a sentence $\Psi_{\Pi,\varphi}$ with the following property. $\Psi_{\Pi,\varphi}$ is not finitely satisfiable if, and only if, $(\Pi, \varphi) \in \text{GVerify}(\text{GF}(\text{HM}), \text{GF}(\mathcal{C}_1))$. Consequently, a finite structure $\mathcal{I}$ satisfies $\Psi_{\Pi,\varphi}$ if, and only if, the computation graph of Π on $\mathcal{I}$ does not satisfy φ.

W.l.o.g., we assume that in a program, distinct rules use distinct variables for the respectively imported elements. Furthermore, we can assume that no guard and no right-hand side of an update rule contains a variable representing an imported element (as all atoms containing such a variable can be replaced by false).

In order to shorten the construction, we first derive the GF-sentence consistent_Π from the program Π. A structure satisfies consistent_Π if, and only if, Π produces a consistent update set on this structure. The definition of the formula is rather straightforward. I.e., it says that if two update rules update the same locations (and both guards are satisfied) then they have to change the content in the same way. E.g., consider two if-clauses **if** g_1 **then** $f(s_1^1, ..., s_n^1) := t_1$ **endif** and **if** g_2 **then** $f(s_1^2, ..., s_n^2) := t_2$ **endif**. This leads to the following formula for consistent_Π (or a conjunct for it):

$$(g_1 \wedge g_2) \rightarrow \left(\left(\bigwedge_{i=1}^{n} s_i^1 = s_i^2 \right) \rightarrow t_1 = t_2 \right)$$

As ASMs from GF($\mathcal{C}_1$) are only allowed to use constant and relation symbols, f has to be a relation symbol.
In the next step, we construct the formula $\Psi^g_{R,\xi}$ for any atomic formula g, any ASM rule $R \in \text{GF}(\mathcal{C}_1)$ (strictly speaking, GF($\mathcal{C}_1$) contains only ASMs but we can immediately generalize it to ASM rules) and any guarded HM-formula ξ. The intended meaning is that g is the guard of R in the currently considered ASM program. Later, the formula $\Psi_{\Pi,\varphi}$ will be defined as $\Psi^{\text{true}}_{\Pi,\varphi}$.
$\Psi^g_{R,\xi}$ is defined by induction.

- If ξ does not contain $\diamond$ (therefore, it is from the guarded fragment of first-order logic and depends only on the first state) then $\Psi^g_{R,\xi} = \xi$.
- $\Psi^g_{\text{Skip},\xi}$ results from ξ by removing all $\diamond$ from ξ.
- If $\xi = \psi_1 \wedge \psi_2$ and R is an arbitrary ASM rule from GF($\mathcal{C}_1$) then $\Psi^g_{R,\xi} = \Psi^g_{R,\psi_1} \wedge \Psi^g_{R,\psi_2}$.
- If $\xi = \psi_1 \vee \psi_2$ and R is an arbitrary ASM rule from GF($\mathcal{C}_1$) then $\Psi^g_{R,\xi} = \Psi^g_{R,\psi_1} \vee \Psi^g_{R,\psi_2}$.

- If $\xi = \neg\psi$ and R is an arbitrary ASM-program from GF($\mathcal{C}_1$) then $\Psi^g_{R,\xi} = \neg\Psi^g_{R,\psi}$.

- If $\xi = \diamond\psi$, where ψ is a guarded HM-formula, and $R =$ **if** α **then** R' **endif** then $\Psi_{R,\xi}$ is the formula

  $$(\text{consistent}_R \rightarrow ((\alpha \rightarrow \Psi_{R',\xi}) \wedge (\neg\alpha \rightarrow \Psi^g_{\text{Skip},\xi})) \wedge (\neg\text{consistent}_R \rightarrow \Psi^g_{\text{Skip},\xi})$$

- If $\xi = \diamond\psi$, where ψ is a guarded HM-formula, and $R = (s := t)$ is an update rule then $\Psi^g_{R,\xi} = \text{apply}(R, \psi, g)$ where $\text{apply}(R, \psi, g)$ is defined by induction on ψ. Essentially, $\text{apply}(R, \psi, g)$ is a formula that is satisfied in $\mathcal{A}$ (together with a variable assignment) if, and only if, ψ is satisfied in a successor of $\mathcal{A}$ (with respect to R). Before giving this definition, some notions are introduced in order to shorten the definition of apply.

  For a boolean-valued term u, $\text{trans}(u, s := t)$ is obtained from u by replacing the relation symbol $R \in \Upsilon$ of u by $R^{s:=t} \notin \Upsilon$.

  Furthermore, for a substitution σ and a tuple of variables $\overline{x} = x_1...x_n$, $\sigma(\overline{x})$ is obtained from $\overline{x}$ by removing from $\sigma(x_1)...\sigma(x_n)$ all constant symbols and all occurrences of a variable which has already occurred before in the tuple $\overline{x}$.

  $\text{apply}(R, \xi, g)$ is defined as follows:

  - if ξ is atomic and not unifiable with s then $\text{apply}(R, \xi, g) = \xi$
  - if ξ is atomic formula and ξ and s are unifiable with most general unifier σ then $\text{apply}(R, \xi, g) = (\text{unchanged}(\xi, s) \rightarrow \xi) \wedge \sigma(t)$
  - $\text{apply}(R, \psi_1 \wedge \psi_2, g) = \text{apply}(R, \psi_1, g) \wedge \text{apply}(R, \psi_2, g)$
  - $\text{apply}(R, \psi_1 \vee \psi_2, g) = \text{apply}(R, \psi_1, g) \vee \text{apply}(R, \psi_2, g)$
  - $\text{apply}(R, \neg\psi, g) = \neg\text{apply}(R, \psi, g)$
  - if $\psi = \forall\overline{x}(\alpha(\overline{x}) \rightarrow \psi')$ then if $\alpha(\overline{x})$ and s are unifiable with most general unifier σ then $\text{apply}(R, \psi, g)$ is the formula

    $$\begin{aligned}&\forall\sigma(\overline{x})\,(\sigma(g) \rightarrow (\text{trans}(\sigma(t), s := t) \rightarrow \text{apply}(R, \sigma(\psi'), g)) \wedge\\ &\forall\overline{x}\,(\sigma(\alpha(\overline{x})) \rightarrow (\neg\sigma(g) \rightarrow \sigma(\psi'))) \wedge\\ &\forall\overline{x}\,(\alpha(\overline{x}) \rightarrow (\text{unchanged}(\alpha(\overline{x}), s) \rightarrow \text{apply}(R, \psi', g)))\end{aligned}$$

    else $\text{apply}(R, \xi, g)$ is the formula

    $$\forall\overline{x}\,(\alpha(\overline{x}) \rightarrow \text{apply}(R, \psi'))$$

– if $\psi = \exists\overline{x}(\alpha(\overline{x}) \wedge \psi')$ then if $\alpha(\overline{x})$ and s are unifiable with most general unifier σ then apply(R, ξ, g) is the formula

$$\begin{aligned}&\exists\sigma(\overline{x})\,(\sigma(g) \wedge (\mathrm{trans}(\sigma(t), s := t) \wedge \mathrm{apply}(R, \sigma(\psi'), g)) \vee\\ &\exists\overline{x}\,(\sigma(\alpha(\overline{x})) \wedge (\neg\sigma(g) \wedge \sigma(\psi'))) \vee\\ &\exists\overline{x}\,(\alpha(\overline{x}) \wedge (\mathrm{unchanged}(\alpha(\overline{x}), s) \wedge \mathrm{apply}(R, \psi', g)))\end{aligned}$$

else apply(R, ξ, g) is the formula

$$\exists\overline{x}(\alpha(\overline{x}) \wedge \mathrm{apply}(R, \psi')))$$

– if $\psi = \diamond\psi'$ then $\mathrm{apply}(\psi) = \mathrm{apply}(R, \Psi_{R,\psi',g})$

where unchanged$(\alpha(\overline{x}), s)$ essentially expresses that $\alpha(\overline{x})$ and s are not unifiable. Formally, this formula is defined as follows.

If $\alpha(\overline{x})$ and s are unifiable then they use the same relation symbol R with arity n. Let $\alpha(\overline{x})$ be of the form $R\overline{y}$ $(\overline{y} = y_1 \dots y_n)$ and s of the form $R\overline{z}$ $(\overline{z} = z_1 \dots z_n)$ where $\overline{y}$ and $\overline{z}$ are n-tuples whose components are variables or constant symbols. A variable or constant symbol is allowed to appear more than once in $\overline{y}$ or $\overline{z}$.

For $i \in \{1, \dots, n\}$ let E_i be the set of indexes j such that the j-th component of $\overline{z}$ is the same as its i-th component. Furthermore, let C be the set of indexes $j \in \{1, \dots, n\}$ such that the j-th component of $\overline{z}$ is a constant symbol.

We define unchanged$(\alpha(\overline{x}), s)$ to be the following formula.

$$\neg\left(\bigwedge_{i\in\{1,\dots,n\}}\bigwedge_{j\in E_i} y_i = y_j \wedge \bigwedge_{i\in C} y_i = z_i\right)$$

If the update rule $s := t$ does not appear inside a parallel composition then replace additionally every appearance of $R^{s:=t} \notin \Upsilon$ by $R \in \Upsilon$.

- If ξ is an arbitrary HM-formula, and $R =$ **do-in-parallel** R_1 R_2 **enddo** then $\Psi^g_{R,\xi} = \left(\mathrm{consistent}_R \to \Psi^g_{R_2,\Psi^g_{R_1,\xi}}\right) \wedge \left((\neg\mathrm{consistent}_R) \to \Psi^g_{\mathrm{Skip},\xi}\right)$.

  If this parallel composition is the outermost one then replace additionally every $R^{s:=t} \notin \Upsilon$ by $R \in \Upsilon$.

- If $\xi = \diamond\psi$ and $R =$ **forall** $\overline{x}$: $\beta(\overline{x}, \overline{y})$ **do** R' **endforall** then $\Psi^g_{R,\xi} = \left(\mathrm{consistent}_R \to \forall\overline{x}\left(\beta(\overline{x}) \to \Psi^\beta_{R',\xi}\right)\right) \wedge \left((\neg\mathrm{consistent}_R) \to \Psi^g_{\mathrm{Skip},\xi}\right)$

- If $\xi = \diamond\psi$ and $R =$ **choose** $\overline{x} : \beta(\overline{z})$ **do** R' **endchoose** then $\Psi^g_{R,\xi}$ is the formula

$$\begin{aligned}&\left(\exists\overline{x}\left(\beta(\overline{x}) \wedge \text{consistent}_{R[\overline{z}/\overline{x}]} \wedge \Psi^{\beta}_{R'[\overline{z}/\overline{x}],\xi}\right)\right) \vee\\ &\left((\forall\overline{x}\neg\beta(\overline{x})) \wedge \Psi^{\beta}_{\text{Skip},\xi}\right) \vee\\ &\left(\left(\forall\overline{x}\left(\beta(\overline{x}) \rightarrow \neg\text{consistent}_{R[\overline{z}/\overline{x}]}\right)\right) \wedge \Psi^{g}_{\text{Skip},\xi}\right)\end{aligned}$$

- If $\xi = \diamond\psi$, $R =$ **import** v R' **endimport** and $\psi = \exists\overline{x}\left(\alpha(\overline{x},\overline{y}) \wedge \tilde{\psi}\right)$ then let trans(R) result from R by replacing all updates containing v by Skip and $\Psi_{R,\xi}$ is the formula

$$\left(\text{consistent}_R \rightarrow \left(\Psi_{\xi,\text{trans}(R)} \vee \bigvee_{\substack{\overline{z} \text{ is a subtuple}\\ \text{of } \overline{x},\overline{z}=x_{i_1}\ldots x_{i_l}}} \exists\overline{z}\Psi_{\alpha\wedge\psi,R}[x_j/v : \bigwedge_{k=1}^{l} j \neq i_k]\right)\right) \wedge$$

$$(\neg\text{consistent}_R \rightarrow \Psi_{\text{Skip},\xi})$$

  where α is the guard of ψ and β is the guard of R.

  The idea of this construction step is the following. The input structure or the initial state does not contain the imported element (it is only contained in the reserve of the corresponding initial structure). Therefore, the proposition of the original formula can be splitted into two propositions: one about tuples not containing the imported element and another one about tuples containing it in at least one component. The imported element might appear more than once in a tuple of elements. All possibilities have to be considered.

The construction steps for the remaining cases result from the duality relations for $\vee$ and $\wedge$, $\exists$ and $\forall$, $\diamond$ and $\Box$.

We define $\Psi_{\Pi,\varphi}$ to be the formula $\Psi^{\text{true}}_{\Pi,\varphi}$. Because φ is from the guarded fragment of HM and Π satisfies the guardedness conditions for ASM programs, the constructed formula is in the guarded fragment of first-order logic (this can be easily proven by induction).

Because the finite satisfiability problem for the guarded fragment of first-order logic is decidable (see e.g. [13]), GVerify(GF(HM), GF($\mathcal{C}_1$)) is decidable. □

The size of the constructed formula is exponential in the size of the input. The finite satisfiability problem for GF is 2EXPTIME-complete and EXPTIME-complete for bounded arity. Therefore, GVerify(GF(HM), GF($\mathcal{C}_1$)) is in 3EXPTIME and in 2EXPTIME for bounded arity.

4.2.2 First-Order CTL

Definition 4.3. Let Υ be a vocabulary containing only relation and constant symbols, and let $\mathcal{A}, \mathcal{B}$ be a Υ-structures, v_1 be an element from the domain of $\mathcal{A}$ and v_2 be an element from the domain of $\mathcal{B}$.

$v_1 \sim_i v_2$ holds if, and only if, for all FO-formulae $\varphi(x)$ over Υ with quantifier depth $\leq i$ (possibly containing equality, $\text{free}(\varphi) = \{x\}$) the following holds:

$$\mathcal{A} \models \varphi[x/v_1] \text{ if, and only if, } \mathcal{B} \models \varphi[x/v_2].$$

Definition 4.4. Let $\Pi \in \mathcal{C}_5, i \in \mathbb{N}$. Let $\mathcal{A}$ and $\mathcal{B}$ be labels in $C_\Pi(\mathcal{I})$. Then $\mathcal{A} \sim_{\mathcal{I},i} \mathcal{B}$ if, and only if, the following conditions are satisfied.

$\mathcal{A}|_{\text{ad}(\mathcal{I})} = \mathcal{A}|_{\text{ad}(\mathcal{I})}$ and there exists a relation $P \subseteq (\text{ad}(\mathcal{A}) - \text{reserve}(\mathcal{I})) \times (\text{ad}(\mathcal{B}) - \text{reserve}(\mathcal{I}))$ such that Pab holds if, and only if, $a \sim_i b$, the projection of P to $\mathcal{A}$ is equal to $\text{ad}(\mathcal{A}) - \text{reserve}(\mathcal{I})$ and the projection of P to $\mathcal{B}$ is equal to $\text{ad}(\mathcal{B}) - \text{reserve}(\mathcal{I})$.

The only possibility to expand the active domain in a run of an ASM is the execution of an import rule. Let us consider those steps in which the atomic type of an imported element can be changed.

If an imported element e is the content of a constant then the atomic type of e can only be changed as long as e is the content of this constant or another one (an update of the form $c := d$ can set the content of c to e - assumed that e is the content of d) and in the step where it is imported. If, at any moment, an imported element is not the content of a constant then it will never be the content of a constant in the future again as it not accessible except that it is left in the reserve and imported again.

Note that the previous argumentation does not hold in the case where functions of arity ≥ 1 are allowed. For example, consider an ASM Π whose vocabulary contains a unary function symbol succ and constant symbols max and min. There is a class of initial states in which succ is interpreted by a succ function and min resp. max by its minimal resp. maximal element.
The iterated execution of the rule

import v
  succ(max) := v
  max := v
endimport

extends the successor function in every step by a new element that becomes its maximal element. Consequently, for every state $\mathcal{A}$ and every element e from the active domain of $\mathcal{A}$ there exists a number $n \in \mathbb{N}$ such that $e = \text{succ}^n(\text{min})$. Therefore, we can access every element of the active domain.

In order to simplify the following proofs, we give the definition of quantifier depth of a CTL-FO formula and an ASM rule in advance.

Definition 4.5. The quantifier depth $\mathrm{qd}(\varphi)$ of a formula $\varphi \in$ CTL-FO is defined by induction on φ.

- if φ is an atomic formula then $\mathrm{qd}(\varphi) = 0$
- $\mathrm{qd}(\psi_1 \vee \psi_2) = \max\{\mathrm{qd}(\psi_1), \mathrm{qd}(\psi_2)\}$
- $\mathrm{qd}(\psi_1 \wedge \psi_2) = \max\{\mathrm{qd}(\psi_1), \mathrm{qd}(\psi_2)\}$
- $\mathrm{qd}(\neg\psi) = \mathrm{qd}(\psi)$
- $\mathrm{qd}(\exists\overline{x}\ \psi) = \mathrm{qd}(\psi) + 1$
- $\mathrm{qd}(\forall\overline{x}\ \psi) = \mathrm{qd}(\psi) + 1$
- $\mathrm{qd}(\mathrm{E}\ \psi_1\ \ \mathrm{U}\ \ \psi_2) = \max\{\mathrm{qd}(\psi_1), \mathrm{qd}(\psi_2)\}$
- $\mathrm{qd}(\mathrm{A}\ \psi_1\ \ \mathrm{U}\ \ \psi_2) = \max\{\mathrm{qd}(\psi_1), \mathrm{qd}(\psi_2)\}$

As CTL-FO is an extension of FO, the above provides also the definition of quantifier depth for FO-formulae.

Definition 4.6. The quantifier depth $\mathrm{qd}(R)$ of an ASM rule R is defined by induction on R.

- $\mathrm{qd}(\textbf{Skip}) = 0$
- if R is an update rule then $\mathrm{qd}(R) = 0$
- $\mathrm{qd}(\textbf{if } \varphi \textbf{ then } R' \textbf{ endif}) = \max\{\mathrm{qd}(\varphi), \mathrm{qd}(R')\}$
- $\mathrm{qd}(\textbf{do-in-parallel } R_1\ R_2 \textbf{ enddo}) = \max\{\mathrm{qd}(R_1), \mathrm{qd}(R_2)\}$
- $\mathrm{qd}(\textbf{forall } x_1 \ldots x_n : \varphi \textbf{ do } R' \textbf{ endforall}) = \max\{\mathrm{qd}(\varphi), \mathrm{qd}(R')\} + n$
- $\mathrm{qd}(\textbf{choose } x_1 \ldots x_n : \varphi \textbf{ do } R' \textbf{ endchoose}) = \max\{\mathrm{qd}(\varphi), \mathrm{qd}(R')\} + n$
- $\mathrm{qd}(\textbf{import } v\ R' \textbf{ endimport}) = \mathrm{qd}(R') + 1$

Essentially, **forall**, **choose** and **import** are also considered as quantifiers.

In the next theorem, we consider the fragment of CTL-Fo that uses only relation and constant symbols.

Definition 4.7. By $\text{CTL-FO}_{\text{rel}}$ we denote the subset of CTL-FO that contains all formulae built over a vocabulary containing only relation and constant symbols.

Theorem 4.8. RVerify($\mathcal{C}_5$, CTL-FO$_{\text{rel}}$) is decidable.

Proof. The main point in the proof is that only a limited number of imported elements (and therefore, a limited number of steps) has an impact on whether a temporal formula is satisfied or not. The limit depends on the quantifier depth of the formula and of the guards in the program.

Let $(\Pi, \mathcal{I}, \varphi)$ be a triple where $\Pi \in \mathcal{C}_5$, $\mathcal{I}$ is an initial structure of Π and $\varphi \in \text{CTL} - \text{FO}_{\text{rel}}$. Further, let n be the maximum of the quantifier depth of φ and the maximal quantifier depth of the guards in Π.

As the active domain of $\mathcal{A}$ is finite, there exist only finitely many Υ-structures over ad($\mathcal{A}$) (where Υ is the vocabulary of $\mathcal{A}$). Further, two imported elements can only appear both in a tuple for which a relation R evaluates to true if they have been imported during the same step or if one of them is stored in a constant or a variable while the other one is imported. Consequently, there are only finitely many equivalence classes of $\sim_{\mathcal{I},n}$ reachable from $\mathcal{I}$ by use of Π.

For the question whether an FO-formula (without equality) is satisfied in a state of a computation it does not matter whether there are k imported elements of a $\sim_{\mathcal{I},n}$-equivalence class or more if $k > n$. It is only important to which $\sim_{\mathcal{I},n}$-equivalence class a structure belongs.

φ is satisfied in $C_\Pi(\mathcal{I})$ if, and only if, φ is satisfied in the labeled transition system $\mathcal{T}$ where

- the states of the transition system are labeled by $\sim_{\mathcal{I},n}$-equivalence classes of $C_\Pi(\mathcal{I})$
- two different states do not have the same labels
- There is an edge between two states u and v if, and only if, there are $\mathcal{A} \in \text{label}(u)$ and $\mathcal{B} \in \text{label}(v)$ such that $\mathcal{A} \vdash_\Pi \mathcal{B}$.

This transition system is finite as there are only finitely many $\sim_{\mathcal{I},n}$-equivalence classes. Furthermore, it can be constructed within finite time as unreachable states respectively their equivalence classes do not need to be considered. Consequently, we only have to check whether the formula holds in this (finite) model (where a formula holds in an equivalence class if, and only if, it holds in one of its elements (in the case of quantifier depth $\leq n$ this is equivalent to the fact that the formula holds in all elements)). □

The undecidability of GVerify($\mathcal{C}_5$, (CTL-FO)$_{\text{rel}}$) is a direct consequence of the undecidability of the finite satisfiability problem for first-order logic. But if we restrict again to the guarded fragments then we obtain a decidable instance of the general verification problem.

Theorem 4.9. GVerify(GF($\mathcal{C}_5$), GF(CTL-FO)) is decidable.

Proof. This proof is essentially a reduction of $\mathrm{GVerify}(\mathrm{GF}(\mathcal{C}_5), \mathrm{GF}(\text{CTL-FO}))$ to $\mathrm{GVerify}(\mathrm{GF}(\mathcal{C}_1), \mathrm{GF}(\mathrm{HM}))$. From theorem 4.2, we then obtain the decidability of $\mathrm{GVerify}(\mathrm{GF}(\mathcal{C}_5), \mathrm{GF}(\text{CTL-FO}))$.

The main point in this proof is the calculation of a number $n(\Pi, \varphi)$ such that

$$(\Pi, \varphi) \in \mathrm{GVerify}(\mathrm{GF}(\mathcal{C}_5), \mathrm{GF}(\text{CTL-FO}))$$

$$\text{if, and only if,}$$

$$\left(\Pi, \bigvee_{0 \leq i \leq n(\Pi,\varphi)} \left(\left(\bigwedge_{0 \leq j < i} \Diamond^j \psi_1\right) \wedge \Diamond^i \psi_2\right)\right) \in \mathrm{GVerify}(\mathrm{GF}(\mathcal{C}_1), \mathrm{GF}(\mathrm{HM}))$$

where $n(\Pi, \varphi)$ depends only on Π and φ. In the special case that the formula has the form $\mathrm{EF}\psi$, we can formulate it much more intuitively

$$(\Pi, \mathrm{EF}\psi) \in \mathrm{GVerify}(\mathrm{GF}(\mathcal{C}_5), \mathrm{GF}(\text{CTL-FO}))$$

$$\text{if, and only if,}$$

$$\left(\Pi, \Diamond^{n(\Pi,\psi)}\varphi\right) \in \mathrm{GVerify}(\mathrm{GF}(\mathcal{C}_1), \mathrm{GF}(\mathrm{HM}))$$

$n(\Pi, \varphi)$ is an upper bound on the number of steps in Π that have to be considered in order to prove or disprove that the computation graph of Π with a distinguished initial state satisfies φ.

Let k be the number of elements maximally imported within one step of the program Π (the number of import rules in Π is an upper bound for k), $\mathrm{qd} := \max\{\mathrm{qd}(\varphi), \mathrm{qd}(\Pi)\}$, and #constants be the number of constants appearing in Π. We define $n(\Pi, \varphi)$ to be

$$2^{\mathrm{qd} \cdot \prod_{R \in \Upsilon} (\#\mathrm{constants}+k)^{\mathrm{arity}(R)}}$$

The left-hand side (and the right-hand side) of an update rule can only be an atom consisting of a relation symbol, constant symbols, and variables bounded by import. Therefore, the locations, whose content can actually be changed, satisfy the following three conditions.

1. The relation symbol of the location has to appear on the left-hand side of an update rule in the program.

2. The elements in the tuple of the location have to be either the content of a constant or an element from the reserve imported by the program.

3. The corresponding atom has to be unifiable with the left-hand side of an update rule in the program.

This indicates that $2^{\mathrm{qd} \cdot \prod_{R \in \Upsilon} (\#\mathrm{constants}+k)^{\mathrm{arity}(R)}}$ is indeed an upper bound on the number of steps for the following reasons. First, there are at most $\prod_{R \in \Upsilon} (\#\mathrm{constants}+k)^{\mathrm{arity}(R)}$ possible atoms on the left-hand side of an update rule and there are at most $2^{\prod_{R \in \Upsilon} (\#\mathrm{constants}+k)^{\mathrm{arity}(R)}}$ possible valuations of the atoms. The maximal

number of distinguishable elements corresponds to the maximal nesting depth. Consequently, $2^{\text{nd}\cdot\prod_{R\in\Upsilon}(\#\text{constants}+k)^{\text{arity}(R)}}$ steps of the ASM suffice in order to prove or disprove that the formula holds in the computation graph generated by the ASM program on a given initial structure □

4.3 Undecidability Results

As already demonstrated in the previous sections, there is a number of decidable instances of the restricted and the general verification problem but there are also numerous undecidable ones. This is not surprising as ASMs can easily simulate all common computation models. In this section, we present some undecidability results.

It is well-known that the question, whether a Turing machine halts on the empty word, is undecidable. We use this fact in a part of the following undecidability proofs.

Before Part I, we have given two examples of an ASM. The second one is a simulation of Turing machine by an ASM. In the remainder of this chapter, we refer to it as the basic simulation of Turing machines by ASMs.

Lemma 4.10. $\text{RVerify}(\mathcal{C}_4, \text{EF})$ is undecidable.

Proof. We prove this lemma via a reduction to the halting problem for Turing machines on the empty word.

Given a Turing machine T, translate it into an equivalent ASM Π_T according to definition 5. As initial state $\mathcal{I}$ choose the one encoding the empty word. The formula to check is EF¬Mode. The Turing machine T halts on the empty word if, and only if, $(\Pi_T, \mathcal{I}, \text{EF}(\neg\text{Mode})) \in \text{RVerify}(\mathcal{C}_4, \text{EF})$. □

Lemma 4.11. $\text{GVerify}(\mathcal{C}_4, \text{EF})$ is undecidable.

Proof. This is a direct consequence of the undecidability of the finite satisfiability problem of first-order logic.

Let $\varphi \in \text{FO}$ not using the nullary relation symbol P. φ is satisfiable if, and only if, $(\textbf{if } \neg\varphi \textbf{ then } P := \text{true } \textbf{endif}, \Box P) \notin \text{GVerify}(\mathcal{C}_4, \text{EF})$. □

We can even strengthen the above lemma by restricting our considerations to the guarded fragment of $\mathcal{C}_4$ and EF.

Lemma 4.12. $\text{GVerify}(\text{GF}(\mathcal{C}_4), \text{GF}(\text{EF}))$ is undecidable.

Proof. The basic simulation of Turing machines (see definition 5) can be modified such that we obtain an ASM from the class $\text{GF}(\mathcal{C}_4)$.

Every function $f/n \in \Upsilon$ of arity n except the constants from Σ or Q is replaced by a relation $f/(n+1) \notin \Upsilon$ of arity $n+1$. The intended meaning is that in the

original simulation $f(\overline{x}) = y$ if, and only if, $f(\overline{x}, y)$ holds in the corresponding state of the modified simulation. E.g., instead of a unary successor function, we use a binary successor relation succ with the intended interpretation that $\mathrm{succ}(x, y)$ holds if, and only if, y is the successor of x.

The rules simulating the work of the Turing machine have to be modified such that they simulate the Turing machine on the modified states and satisfy the conditions for $\mathrm{GF}(\mathcal{C}_4)$.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing machine. The simulating ASM is defined by the program

forall $x : \mathrm{head}(x)$ **do**
  $\{R_{aq}\}_{(a,q)\in\Sigma\times Q}$
endforall

where $\{R_{aq}\}_{(a,q)\in\Sigma\times Q}$ is the parallel composition of the rules $R_{aq}, (a, q) \in \Sigma \times Q$.

For $(a, q) \in \Sigma \times Q$ with $\delta(a, q) = (p, b, -1)$ the rule R_{aq} is defined as follows.

if $\mathrm{content}(x, a) \wedge \mathrm{state}(q)$ **then**
  do-in-parallel
    $\mathrm{content}(x, b) := \mathrm{true}$
    $\mathrm{state}(p) := \mathrm{true}$
    $\{\mathrm{inscr}(x, c) := \mathrm{false}\}_{c\in\Gamma-\{b\}}$
    $\{\mathrm{state}(p') := \mathrm{false}\}_{p'\in Q-\{p\}}$
    forall $y : \mathrm{pred}(x, y)$ **do**
      $\mathrm{head}(x) := \mathrm{false}$
      $\mathrm{head}(y) := \mathrm{true}$
    endforall
  enddo
endif

For $(a, q) \in \Sigma \times Q$ with $\delta(a, q) = (p, b, 1)$ the rule R_{aq} is defined as follows.

if $\mathrm{content}(x, a) \wedge \mathrm{state}(q)$ **then**
  do-in-parallel
    $\mathrm{content}(x, b) := \mathrm{true}$
    $\mathrm{state}(p) := \mathrm{true}$
    $\{\mathrm{inscr}(x, c) := \mathrm{false}\}_{c\in\Gamma-\{b\}}$
    $\{\mathrm{state}(p') := \mathrm{false}\}_{p'\in Q-\{p\}}$
    if $\mathrm{head}(x) \wedge \mathrm{max}(x)$ **then**
      import v
        $\mathrm{max}(x) := \mathrm{false}$
        $\mathrm{succ}(x, v) := \mathrm{true}$
        $\mathrm{pred}(v, x) := \mathrm{true}$

```
            max(v) := true
            max(x) := false
            head(v) := true
        endimport
        else
          forall y : succ(x, y) do
            head(x) := false
            head(y) := true
          endforall
        endif
      enddo
    endif
```

For $(a, q) \in \Sigma \times Q$ with $\delta(a, q) = (p, b, 0)$ the rule R_{aq} is defined as follows.

```
if content(x, a) ∧ state(q) then
  do-in-parallel
    content(x, b) := true
    state(p) := true
    {inscr(x, c) := false}_{c∈Γ−{b}}
    {state(p′) := false}_{p′∈Q−{p}}
  enddo
endif
```

□

Note that the use of **import** is not excluded. Without **import**, we obtain finite state systems resp. finite computation graphs. In the case of finite computation graphs, the restricted verification problem is obviously decidable.

On first view, it might seem surprising that RVerify($\mathcal{C}_3$, CTL-FO$_{\text{rel}}$) is undecidable as the respective instance of the restricted verification problem is decidable for $\mathcal{C}_5$. One might think that, as for $\mathcal{C}_5$, it suffices to consider the equivalence classes. If two imported elements are equivalent after the later imported one is imported then they will be equivalent in all states reachable from this one. The use of forall does not destroy the property because the guard of a forall-rule is satisfied by all elements of an $\sim$-equivalence or by none. Therefore, the rule inside a forall-rule is applied to all or to none of the elements in an equivalence class. Until here, the argumentation is correct but this does not imply that one can reduce an instance of RVerify($\mathcal{C}_3$, CTL-FO) to a finite transition system with the question whether the formula holds.

The reason is that by use of **forall**, it is possible to change the atomic type of an imported element after it has been imported and it is not anymore the content of a constant, not only in parallel (at the same time it is imported).

Lemma 4.13. RVerify($\mathcal{C}_3$, EF) is undecidable.

Proof. We prove this lemma via a reduction of RVerify($\mathcal{C}_4$, EF).

The only thing possible in $\mathcal{C}_4$ but not in $\mathcal{C}_3$ is the use of function symbols. By use of function symbols, we can construct nested terms. These can occur in different contexts, namely

1. as left-hand side of an update-rule or right-hand side (update-rules of the form $s := t$ are possible where s and t are such terms)

2. as an argument in a boolean-valued term (i.e., if R a relation symbol then $R...t...$ is possible).

The program considered in the proof of 4.11 has to be modified in the following way:

1. If Π contains only constant and relation symbols then trans(Π) := Π.

2. Let Π contain a term of the form $s = f(t_1, ..., t_r)$, where f is a function symbol, and let z be a variable not used in Π. Then trans(Π) is equal to trans applied to the following rule:

   forall $z : Ft_1...t_rz$ **do**
   $\Pi[s/z]$
   endforall

Instead of φ we have to verify the formula

$$(\bigwedge_{f \in \Upsilon_{\text{function}}} (\forall \overline{x} \forall y \; R\overline{x} \rightarrow \forall z (R\overline{x}z \rightarrow z = y))) \rightarrow \text{trans}(\varphi)$$

where trans(φ) is defined by induction on φ as follows.

1. If φ only contains relation and constant symbols (and therefore, no function symbols of arity ≥ 1) then trans(φ) = φ.

2. Let φ contain a term of the form $s = f(t_1, ...t_r)$ and let z be a variable not used in φ. Then trans(φ) = trans($\exists z(F(t_1, ..., t_r) \wedge \varphi[s/z])$)

□

Analogously to lemma 4.13, we can prove the following lemma.

Lemma 4.14. RVerify($\mathcal{C}_3$, EF) is undecidable.

Proof. In the proof of lemma 4.13, we can replace every occurrence of **forall** by **choose**. This means, every forall-rule is replaced by an element-nondeterministic rule with the same guard. With the same arguments, we obtain a prove for lemma 4.14. □

Though a number of interesting properties is expressible in EF, it is one of the smallest fragments of temporal first-order logic above Hennessy-Milner first-order logic. As we obtain undecidability results already for EF, it does not make sense to consider fragments beyond EF with respect to decidability of the verification problem.

4.4 Remarks and Conclusion

Another way to obtain infinite state spaces resp. infinite computation graphs is to allow for a state an infinite static substructure instead of using **import**. To be more precise, the vocabulary Υ is partitioned into two disjoint sets Υ_{stat} and Υ_{dyn}. Υ_{stat} contains all static relation and function symbols (whose interpretation is never changed during a computation) and Υ_{dyn} contains all dynamic relation and function names. The active domain of the reduct of a state to Υ_{dyn} has to be finite but the active domain of the reduct of a state to Υ_{stat} is allowed to be infinite. Note that it is not required that the two subsets of the universe are disjoint.

But this leads rather directly to undecidability of the restricted verification problem. E.g. choose $\Upsilon_{\text{stat}} = \{\text{Succ}, \text{Pred}\}$ and $\mathbb{N}$ as the active domain of the reduct of a state to Υ_{stat} where Succ and Pred are interpreted as the usual successor and predecessor function on $\mathbb{N}$. Turing machines can be simulated as before but without using **import**. Obviously, it is not necessary to import elements.

In this section, we have considered simple restrictions on ASMs and logics in order to obtain decidable instances of the verification problem.

One of the ideas for identifying verifiable classes of ASMs and formalisms is to forbid certain advanced construction elements or restricting their use in a rather simple way. Advanced construction elements are those which allow rules of the form **forall** $\overline{x} : \varphi(\overline{x})$ **do** Π **endforall**, the choice of an element from the domain of a state or the use of import.

It turned out that the restricted verification problem for relational branching time first-order logic and the class $\mathcal{C}_5$ of ASMs is decidable. Decidability for the general verification problem is obtained by e.g. restricting to the guarded fragments.

If we restrict the specification logic to the first-order variant of Hennessy-Milner logic then we do not need to put any restriction on the ASMs in order to obtain

decidability for the restricted verification problem. Already the restriction to the guarded fragment leads to decidability in the case of the general verification problem.

These considerations demonstrate that simple restrictions lead to decidability results in a few cases only.

The first-order variant of Hennessy-Milner logic (HM-FO) is a rather weak logic. A given formula only allows propositions about a fixed number of steps of an ASM (bounded by the maximal number of nested $\Diamond$). However, there is no uniform bound for all formulae and consequently, HM-FO may already be strong enough for some applications.

On the other hand, the class $\mathcal{C}_5$ of ASMs is not very expressive. as it is not possible to update the content of locations with elements that are not from the reserve (addressable via import) and that are not the content of a constant. Therefore, all locations of the input state containing at least one such element are static.

In order to obtain further decidability results, we have to consider classes that result from other kinds of restrictions. Some restrictions have already been considered in [30] and [32]. We briefly summarize these results in section 5. Afterwards, we introduce new classes of ASMs and fragments of temporal logic for which we can prove decidability of the verification problem.

5 Known Advanced Decidability Results

5.1 Nullary Programs

In [30], M. Spielmann has defined the class of nullary (ASM-)programs. A nullary program is a nondeterministic basic ASM program where every dynamic function is nullary. "Basic" means the following in this context:

- the guards in the if-clauses and the choose-rules are not allowed to contain quantifiers
- the use of **import** or **forall** is not allowed

M. Spielmann has proven that the general verification problem for the existential and the universal fragment of the fragment of a CTL*-like logic containing only state formulae is decidable. The proof is done via a reduction to the satisfiability problem of existential transitive closure logic.

5.2 ASM transducers

In this section, we give a short introduction to ASM transducers. This one is more detailed than the one to nullary ASMs as we will compare our results to the ones on ASM transducers later in this part.

In [2], relational transducers have been introduced. Informally, a relational transducer computes for a sequence of input relations a sequence of output relations. To be more precise, a state of a relational transducer is a relational first-order structure consisting of a (dynamic) memory part and a (static) database part. In every computation step, the transducer receives the input relations from the environment. It reacts by producing output relations and updating its memory.

The focus of [32] is on a class of relational transducers that is defined via a special class of ASMs, so-called ASM (relational) transducers.

Definition 5.1. A transducer vocabulary is defined to be a quintuple of the form$(\Upsilon_{in}, \Upsilon_{db}, \Upsilon_{mem}, \Upsilon_{out}, \Upsilon_{log})$ whose components finite relational vocabularies where the first four vocabularies are pairwise disjoint and $\Upsilon_{log} \subseteq \Upsilon_{in} \cup \Upsilon_{out}$. Let

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

$$\textbf{if } \varphi(\overline{x}) \textbf{ then } (\neg)R(\overline{x})$$

where $\varphi(\overline{x})$ (the guard of the rule) is an FO-formula over $\Upsilon_{in} \cup \Upsilon_{db} \cup \Upsilon_{mem}$ with $\text{free}(\varphi) = \{\overline{x}\}$, and $R(\overline{x})$ is an atomic FO formula over $\Upsilon_{mem} \cup \Upsilon_{out}$ which occurs positively on the right-hand side of the rule if $R \in \Upsilon_{out}$.

Semantics of ASM transducer programs. The relational transducer $T_{(\Upsilon,\Pi)}$ over Υ defined by Π is a function mapping every state S over Υ t a finite structure $T_{(\Upsilon,\Pi)}(S)$ over $\Upsilon_{mem} \cup \Upsilon_{out}$. The (new) input component if S' will be provided by the environment. The database component is copied from S.

The following assumptions are made about Π.

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

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

$$\begin{array}{lcl}
\varphi_R(\overline{x}) & := & \bigvee\{\varphi(\overline{x}) : (\textbf{if } \varphi(\overline{x}) \textbf{ then } R(\overline{x})) \text{ is a rule in } \Pi\} \\
\psi_R(\overline{x}) & := & \bigvee\{\varphi(\overline{x}) : (\textbf{if } \varphi(\overline{x}) \textbf{ then } \neg R(\overline{x})) \text{ is a rule in } \Pi\} \\
\chi_R(\overline{x}) & := & (\varphi_R(\overline{x}) \wedge \neg\psi_R(\overline{x}))\vee \\
 & & (\varphi_R(\overline{x}) \wedge \psi_R(\overline{x}) \wedge R(\overline{x})))\vee \\
 & & (\neg\varphi_R(\overline{x}) \wedge \neg\psi_R(\overline{x}) \wedge R(\overline{x})))
\end{array}$$

where $\bigvee \emptyset \equiv \text{false}$.

For every state S over Υ, let the finite structure $T_{(\Upsilon,\Pi)}(S)$ over $\Upsilon_{mem} \cup \Upsilon_{out}$ be defined as follows:

- the universe of $T_{(\Upsilon,\Pi)}(S)$ is that of S,
- for every $R \in \Upsilon_{mem}$, $R^{T_{(\Upsilon,\Pi)}(S)} := \chi_R^S$, and
- for every $R \in \Upsilon_{out}$, $R^{T_{(\Upsilon,\Pi)}(S)} := \varphi_R^S$,

where χ_R^S and φ_R^S denote the answer relations of the queries χ_R and φ_R on S, respectively.

Definition 5.2. An ASM (relational) transducer T is a pair (Υ, Π) consisting of a transducer vocabulary Υ and an ASM transducer program Π over Υ.

Runs. Let T be an ASM transducer over Υ. A database D appropriate for T is a finite structure over Υ_{db}. An input sequence $\overline{I}$ appropriate for T and D is an infinite sequence of finite structures over Υ_{in} where each I_i has the same universe as D.

Let D be a database and $\overline{I}$ an input sequence, both appropriate for T (and D). The run ρ of T on D and $\overline{I}$ is an infinite sequence$(S_i)_{i\in\omega}$ of states over Υ uniquely determined by the following conditions. For every $i \in \omega$,

- $S_i|_{\Upsilon_{in}} = I_i$,
- $S_i|_{\Upsilon_{db}} = D$, and
- $S_i|_{\Upsilon_{mem}\cup\Upsilon_{out}} = (D, \emptyset)$ if $i = 0$ (i.e., all memory and output relations are empty in the initial state S_0); otherwise $S_i|_{\Upsilon_{mem}\cup\Upsilon_{out}} = T(S_{i-1})$.

To prove the decidability of

- the problem of the verifying of temporal properties of ASM (relational) transducers
- log equivalence of two ASM (relational) transducers
- finite log validation

for ASM (relational) transducers, the following restrictions have been introduced:

1. The database on which a relational transducer is supposed to run is provided and fixed during verification.
2. The maximal input flow (Let ρ be a run of an ASM transducer over Υ. The maximal input flow of ρ is the maximal natural number in $\{|R^{S_i}| : R \in \Upsilon_{in}, i \in \omega\}$ where $|R^{S_i}|$ denotes the cardinality of the input relation R^{S_i}.) to which a relational transducer is exposed is a priori bounded.
3. The arities of the relations are bounded a priori.
4. The logic for verifying temporal properties is not full first-order temporal logic; it is not allowed to quantify existentially into temporal contexts, only universally (the fragment is called UT).

The general verification problem for the logic UT has been proven to be decidable in the following cases:

1. The first and the third restriction are satisfied.
2. The first and the second restriction are satisfied.

3. The second restriction is satisfied and only the guarded fragment of UT is allowed and the guards in the ASM (relational) formula are in the guarded fragment of first-order logic

The same cases have been proved to be decidable for the other problems. The complexity of the verification problem depends on the respective restrictions.

The restrictions in [32] are rather strong. A fixed database is not suitable if the database of a relational transducer is not fixed and is changed after a certain time. For being sure that the transducer also has the desired temporal properties also with the new database, one has to verify it again with the new database. In order to witness that such cases really appear in practice, we consider an example from the area of commerce. The database of a transducer modeling a commerce system might correspond to a catalogue giving the prices of the products. Usually, the products do not change often but the prices do. Consider for example someone selling newspapers. A certain set of newspapers is offered and this set normally does not changed. But the prices do increase rather often.

Limiting the input flow a priori is also problematic because in many applications, it is not possible to determine such limits. Later, we will see such an example.

A further point that is not always wanted, is that the memory relations and the output relations are always initialized by the empty set. This means that, in the first state of a run, the output relations and the memory relations are always interpreted by the empty set. If one initializes a transducer and there exists already some memory data, it is not possible to take it over into the new memory data. The only alternative would be to put it into the database (and then copy it into the memory) but then the original memory data is static and can not be changed or deleted from the database. This is not the intention of memory data.

6 A Class of Guarded Abstract State Machines

In this chapter, we introduce a new decidable instance of the general verification problem. The class of ASMs that we consider is a well-defined fragment of GF($\mathcal{D}$).

6.1 Definition and a Normal Form

Definition 6.1. The class ASM_1 contains exactly those ASMs in $\mathcal{D}$ that satisfy the following conditions.

- The vocabulary contains only constant and relation symbols
- The use of import is not allowed
- Update rules have the form $R\bar{c}_1 x\bar{c}_2 x...\bar{c}_n := t$ where $R \in \Upsilon$ is a relation symbol (the left-hand side of an update-rule contains at most one free variable)
- The guards of the if-clauses have to be in the guarded fragment of first-order logic
- Forall-rules have the form **forall** $\overline{x} : \varphi(\overline{x}, \overline{y})$ Π **endforall** where φ is an atomic formula $\text{free}(\Pi) \cup \{\overline{x}\} \subseteq \text{free}(\varphi)$

This class of ASMs is denoted by ASM_1.

An equivalent definition of ASM_1 is the following. ASM_1 is the subset of GF($\mathcal{D}$) containing exactly those ASMs in GF($\mathcal{D}$) whose update rules have the form $R\bar{c}_1 x\bar{c}_2 x...\bar{c}_n := t$ (the left-hand side of an update-rule in an ASM from ASM_1 contains at most one free variable).

Starting from this second, equivalent definition of ASM_1, we obtain directly a proposition on the expressive power of ASM_1. We know from theorem 2.9 that every sentence in μGF is equivalent to a GF($\mathcal{D}$) query and conversely, there exists a GF($\mathcal{D}$) query for which there is no equivalent formula in μGF.

In order to obtain a result on the expressiveness of ASM_1, we just have to restrict the use of the fixed point variables in the formulae such that if an atomic formula involves a fixed point variable R then every occurrence of R has to be of the form $R\bar{c}_1 x\bar{c}_2 x...\bar{c}_n$ (the atomic contains at most one free variable).

The following lemma provides a normal form for ASMs in ASM_1.

Lemma 6.2. For every ASM in ASM_1, we can effectively construct an equivalent ASM in ASM_1 of the form

do-in-parallel
  Π_1
  .
  .
  .
  Π_n
enddo

where each of the programs Π_i, $i \in \{1, .., n\}$ is of the form

forall x : true **do**
  if $\varphi_t^{\beta(x)}$ **then** $\beta(x) :=$ true **endif**
endforall

or of the form

forall x : true **do**
  if $\varphi_f^{\beta(x)}$ **then** $\beta(x) :=$ false **endif**
endforall

where $\varphi_t^{\beta(x)}$ is equal to false if an update of $\beta(x)$ to true is not possible and $\varphi_f^{\beta(x)}$ is equal to false if an update of $\beta(x)$ to false is not possible.

Furthermore, for every atomic formula $\beta(x)$, there appears at most one program of the first form and at most one of the second form.

Proof. We prove this lemma by effectively constructing for an arbitrary ASM in ASM_1 an equivalent ASM in ASM_1 in the described form.

ASMs in ASM_1 can be transformed into a parallel composition of programs that have the form of the following program Π

forall $x, \overline{y} : \alpha(x, \overline{y})$ **do**
  if φ **then** $\beta(x) := \gamma(x, \overline{y})$ **endif**
endforall

with free(φ) $\subseteq \{x, \overline{y}\}$, $\alpha(x, \overline{y})$, $\beta(x)$, $\gamma(x, \overline{y})$ are atomic formulae where $\beta(x)$ is of the form $R\overline{c}_1 x \overline{c}_2 ... \overline{c}_i x \overline{c}_{i+1} ... \overline{c}_n$ (x is a variable and $\overline{c}_i$, $i \in \{1, ..., n\}$, are (possibly empty) tuples of constant symbols from the vocabulary of Π), and free(γ) $\subseteq \{x, \overline{y}\}$.

Π is equivalent to

do-in-parallel
  forall x : true **do**
    if $\exists \overline{y}(\alpha(x, \overline{y}) \wedge \varphi \wedge \gamma(x, \overline{y}))$ **then** $\beta(x) :=$ true **endif**
  endforall
  forall x : true **do**
    if $\exists \overline{y}(\alpha(x, \overline{y}) \wedge \varphi \wedge \neg\gamma(x, \overline{y}))$ **then** $\beta(x) :=$ false **endif**
  endforall
enddo

Furthermore, the program

do-in-parallel
  forall x : true **do**
    if ψ_1 **then** $\beta(x) := v$ **endif**
  endforall
  forall x : true **do**
    if ψ_2 **then** $\beta(x) := v$ **endif**
  endforall
enddo

is equivalent to

forall x : true **do**
  if $\psi_1 \vee \psi_2$ **then** $\beta(x) := v$ **endif**
endforall

($v \in \{\text{true}, \text{false}\}$).

Therefore, for an ASM in ASM_1, we can effectively construct a ASM in ASM_1 such that, for every atomic formula $\beta(x)$, there is exactly one rule

forall x : true **do**
  if $\varphi_t^{\beta(x)}$ **then** $\beta(x) :=$ true **endif**
endforall

and one rule

forall x : true **do**
  if $\varphi_f^{\beta(x)}$ **then** $\beta(x) :=$ false **endif**
endforall

where $\varphi_t^{\beta(x)}$ is equal to false if an update of $\beta(x)$ to true is not possible and $\varphi_f^{\beta(x)}$ is equal to false if an update of $\beta(x)$ to false is not possible.
(The above construction does not exclude that there are a state $\mathcal{A}$ and a tuple $\overline{a}$ of elements from the domain of $\mathcal{A}$ such that a location $(R, \overline{a})$ is accessed by different rules. This is caused by the possibility that two different constant symbols may be interpreted by the same element from the domain of the state. It can be eliminated by adding inequalities to the guards of the if-clauses. E.g. assume that there are two atomic formulae $\beta_1(x) = Rcx$ and $\beta_2(x) = Rdx$ with guards in the if-clauses $\varphi_t^{\beta_1(x)}, \varphi_t^{\beta_2(x)}, \varphi_f^{\beta_1(x)}, \varphi_f^{\beta_2(x)}$. Add $c \neq d$ to the above guards of the if-clauses and add to the program:

forall x : true **do**
  if $((\varphi_f^{\beta_1(x)} \wedge \varphi_t^{\beta_2(x)}) \vee (\varphi_t^{\beta_1(x)} \wedge \varphi_f^{\beta_2(x)})) \wedge c = d$ **then**
    do-in-parallel
      $Rcx :=$ true
      $Rdx :=$ false
    enddo
  endif
endforall

If, instead of Rdx, the atom Rxd would have been appeared then instead of $c = d$ the formula $x = c \wedge x = d$ would have been used.) □

6.2 Logical Background

Remember the definition of the guarded fragment of first-order logic.

The logic TLGF is the guarded fragment of first-order temporal logic defined in [25].

Definition 6.3. By TLGF we denote the guarded fragment of temporal first-order logic, GF(TLFO).

In [25], the fragment $\mathcal{TL}_1$ of TLFO has been defined as follows.

Definition 6.4. Denote by $\mathcal{TL}_1$ the set of all temporal first-order formulae φ such that any subformula of φ of the form ψ_1 U ψ_2 or ψ_1 S ψ_2 has at most one free variable. Such formulae are called monodic. In different words, monodic formulas allow quantification into temporal contexts only with at most free variable.

Definition 6.5. By TLGF_1 we denote $\mathrm{TLGF} \cap \mathcal{TL}_1$.

The fragment TLGF_1 is suitable for describing temporal properties of ASMs in ASM_1. We prove the general verification problem to be decidable for TLGF_1. Many interesting properties are expressible in this formalism. This is true for mainly two reasons.

1. The left-hand sides of update rules in ASMs in ASM_1 contain at most one variable.

2. Many interesting properties can be described by a sentence of the form $\psi_1 \ \mathrm{U} \ \psi_2$ where ψ_1 and ψ_2 are (guarded) first-order sentences. For example, safety properties often have this form. A state is considered as "not save" if, and only if, the sentence ψ is not satisfied (ψ depends on the context). Therefore, the formula to verify (for verifying the safety property) is $G\neg\psi$. In these cases, the possibilities of $\mathcal{TL}_1$ are even not completely exploited.

In [24], the decidability of the satisfiability problem for TLGF_1 in $(\mathbb{N}, <)$ over finite, not expanding domain has been proven. Essentially, it corresponds to the following question.

Given a sentence $\varphi \in \mathrm{TLGF}_1$ (possibly with equality). Is φ satisfiable in $(\mathbb{N}, <)$ over a finite domain?

Definition 6.6. By $\mathrm{FinSat}(\mathrm{TLGF}_1, (\mathbb{N}, <))$, we denote the set of all TLGF_1-sentences that are satisfiable in $(\mathbb{N}, <)$ over finite, not expanding domains.

So, in [24] it has been proven that $\mathrm{FinSat}(\mathrm{TLGF}_1, (\mathbb{N}, <))$ is decidable.

Furthermore, in [25] the undecidability of the corresponding problem for the fragment $\mathcal{TL}_2 \cap \mathrm{TLGF}$ (where $\mathcal{TL}_2$ is defined analogously to $\mathcal{TL}_1$ but allowing two free variables) has been proven (via a reduction to the tiling problem for $\mathbb{N} \times \mathbb{N}$).

6.3 Deciding the General Verification Problem

Definition 6.7. A quasi-atom is a formula of the form $\bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{n} y_{i,j}^1 \neq y_{i,j}^2 \right) \wedge \alpha(\overline{x})$ or of the form $\left(\bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{n} y_{i,j}^1 \neq y_{i,j}^2 \right) \right) \rightarrow \alpha(\overline{x})$ where α is an atomic formula with $\mathrm{free}(\alpha) \subseteq \{\overline{x}\}$ and $y_{i,j}^1, y_{i,j}^2$ ($i \in \{1, ..., m\}, j \in \{1, ..., n\}$) are constant symbols or variables from $\overline{x}$.

Note that every atomic formula is also a quasi-atom.

Definition 6.8. An extended substitution is a finite set of simultaneous replacements of variables and constants by terms.

Definition 6.9. A quasi-atom of the form $\bigwedge_{i=1}^{m}\left(\bigvee_{j=1}^{n} y_{i,j}^1 \neq y_{i,j}^2\right) \wedge \alpha(\overline{x})$ or of the form $\left(\bigwedge_{i=1}^{m}\left(\bigvee_{j=1}^{n} y_{i,j}^1 \neq y_{i,j}^2\right)\right) \rightarrow \alpha(\overline{x})$ is unifiable with an atomic formula $\gamma(\overline{z})$ if α and γ are unifiable if there exists an extended substitution σ such that $\sigma(\alpha)$ and $\sigma(\gamma)$ are identical and $\sigma(\bigwedge_{i=1}^{n} y_i^1 \neq y_i^2)$ is satisfiable.

Lemma 6.10. For every ASM $\Pi \in \mathrm{ASM}_1$, there exists an effectively constructable sentence $\mathrm{consistent}_\Pi \in \mathrm{GF}$ such that for all states $\mathcal{A}$ of Π the following holds:

$\mathcal{A} \models \mathrm{consistent}_\Pi$ if, and only if, the update set produced by Π on $\mathcal{A}$ is consistent.

Proof. The construction of $\mathrm{consistent}_\Pi$ is straightforward. Two rules can only cause a collision if they update the same relation. Therefore, consider every atom appearing on the left-hand side of an update-rule. The formula can be directly gained from the respective guards.

In the remainder of this proof, we give the complete construction of $\mathrm{consistent}_\Pi$.

W.l.o.g., we can assume that Π is in the normal form lemma 6.2. For $v \in \{\mathrm{true}, \mathrm{false}\}$, let L_Π^v be the set of all pairs where the first component is atomic formula appearing as a left-hand side of an update rule in Π whose right-hand side is v and the second component is the guard of the respective if-clause. Furthermore, for every atomic formula α, $v \in \{\mathrm{true}, \mathrm{false}\}$, let $U_\Pi^{\alpha,v} \subseteq L_\Pi^v$ such that the first component of all pairs in $U_\Pi^{\alpha,v}$ is unifiable with α.

For two unifiable atomic formulae α and β, denote by $\sigma_{\alpha,\beta}$ their most general unifier. Then $\mathrm{consistent}_\Pi$ is the formula

$$\forall x\Big(\bigwedge_{(\alpha,\varphi_\alpha)\in L_\Pi^{\mathrm{true}}} \bigwedge_{(\beta,\varphi_\beta)\in U_\Pi^{\alpha,\mathrm{false}}} \sigma_{\alpha,\beta}(\varphi_\alpha) \rightarrow \neg\sigma_{\alpha,\beta}(\varphi_\beta)\Big)$$
$$\wedge\forall x\Big(\bigwedge_{(\alpha,\varphi_\alpha)\in L_\Pi^{\mathrm{false}}} \bigwedge_{(\beta,\varphi_\beta)\in U_\Pi^{\alpha,\mathrm{true}}} \sigma_{\alpha,\beta}(\varphi_\alpha) \rightarrow \neg\sigma_{\alpha,\beta}(\varphi_\beta)\Big)$$

□

Note that $\mathrm{GF} \subseteq \mathrm{TLGF}_1$ and therefore, $\mathrm{consistent}_\Pi \in \mathrm{TLGF}_1$.

Lemma 6.11. For every ASM $\Pi \in \mathrm{ASM}_1$ there exist effectively constructable sentences $\psi_\Pi, \chi_\Pi \in \mathrm{TLGF}_1$ such that for all states $\mathcal{A}, \mathcal{B}$ of Π and all temporal structures $\mathcal{M} = ((\mathbb{N}, <), D, I)$ with $I(1) = \mathcal{A}$, $I(2) = \mathcal{B}$ and the update set produced by Π on $\mathcal{A}$ is consistent, the following holds:

$\mathcal{A} \vdash_\Pi \mathcal{B}$ iff $(\mathcal{M}, 1) \models \psi_\Pi \wedge \chi_\Pi$ and for all quasi-atoms $\alpha(\overline{x})$ not unifiable with the left-hand side of an update-rule in Π $\alpha(\mathcal{A}) = \alpha(\mathcal{B})$

Proof. First, we construct the formula ψ_Π. It formalizes the changes induced by Π and is defined by induction on Π.

- If Π has the form $(s := t)$ then $\psi_\Pi := (\mathrm{X}s) \leftrightarrow t$.
- If Π has the form **if** g **then** Π' **endif** then $\psi_\Pi := g \rightarrow \psi_{\Pi'}$.
- If Π has the form **do-in-parallel** Π_0 Π_1 **enddo** then $\psi_\Pi := \psi_{\Pi_0} \wedge \psi_{\Pi_1}$.
- If Π has the form **forall** $\overline{x}$: $\alpha(\overline{x}, \overline{y})$ **do** $\Pi'(\overline{x}, \overline{y})$ **endforall** then $\psi_\Pi := \forall \overline{x}(\alpha(\overline{x}, \overline{y}) \rightarrow \psi_{\Pi'}(\overline{x}, \overline{y}))$.

ψ_Π is a sentence as Π has no free variables. In ψ_Π, the only temporal operators are the X introduced by an update rule. Therefore, sub-formulae of ψ_Π with a temporal operator outermost are of the form Xs where s is the left-hand side of an update rule in Π. By the definition of ASM_1, s and Xs contain at most one variable. Because of the restriction on the guards of Π, $\psi_\Pi \in \mathrm{TLGF}_1$.

χ_Π avoids changes in those atoms appearing on the left-hand side of an update-rule in Π not indicated by the program.

Assuming, w.l.o.g., that the program Π is in the normal form from lemma 6.2, we now construct the formula χ_Π.

For an arbitrary element a in the domain of the current state, the intended formula χ_Π expresses the following. If there are no assignments to $\overline{y}$ such that φ holds (with a assigned to x), then the interpretation of $\beta(a)$ does not change.

The formula is the conjunction of two formula, each treating one of the following two cases:

1. $\beta(x)$ (resp. with a assigned to x) is interpreted by true in the current state.
2. $\beta(x)$ (resp. with a assigned to x) is interpreted by false in the current state.

The first case is treated by the formula $\forall x(\beta(x) \rightarrow (\neg\varphi_f \rightarrow (X\beta(x))))$.

The second case is treated by the formula $\forall x((\neg\beta(x)) \rightarrow (\neg\varphi_t \rightarrow (X\neg\beta(x))))$.

Both formulae are in $\mathrm{TLGF}_1 = \mathcal{TL}_1 \cap \mathrm{TLGF}$. Obviously, the condition introduced by TLGF_1 is satisfied as the only temporal operator is X which is only used in front of $\beta(x)$ resp. $\neg\beta(x)$ (and $\beta(x)$ contains at most one free variable (namely, x)). Furthermore, the guardedness condition is satisfied because the only quantifications are those appearing in φ_t and φ_f and the ones of the form $\forall x\psi$ (the outermost formulae). Those quantifications appearing in φ_t and φ_f satisfy the guardedness condition as $\varphi_t \in \mathrm{GF}$ and $\varphi_f \in \mathrm{GF}$. The two quantifications of the form $\forall x\psi$ are no problem as there is no restriction on quantifications with just one variable (the guard is assumed to be $x = x$). □

If the update set produced by a program on a structure is not consistent then nothing changes.

Lemma 6.12. For every ASM $\Pi \in \mathrm{ASM}_1$ there exists an effectively constructable sentence $\mathrm{noChanges}_\Pi \in \mathrm{TLGF}_1$ such that for all states $\mathcal{A}, \mathcal{B}$ of Π and for all temporal structures $\mathcal{M} = ((\mathbb{N}, <), D, I)$ with $I(1) = \mathcal{A}$ and $I(2) = \mathcal{B}$, the following holds:

$$(\mathcal{M}, 1) \models \mathrm{noChanges}_\Pi \quad \text{iff} \quad \begin{array}{l}\text{for all atoms } \alpha(\overline{x}) \text{ appearing on the left-hand side} \\ \text{of an update-rule in } \Pi\ \alpha(\mathcal{A}) = \alpha(\mathcal{B})\end{array}$$

Proof. Let $\Pi \in \mathrm{ASM}_1$ and L_Π be the set of all atoms appearing on the left-hand side of an update rule appearing in Π. Then

$$\bigwedge_{\beta(x) \in L_\Pi} \forall x(\beta(x) \leftrightarrow X\ \beta(x))$$

This is a sentence form TLGF_1 as for every element $\beta(x)$ of L_Π $|\mathrm{free}(\beta(x))| \leq 1$ holds. □

Theorem 6.13. $\mathrm{GVerify}(\mathrm{ASM}_1, \mathrm{TLGF}_1)$ is decidable.

Proof. We prove the decidability of $\mathrm{GVerify}(\mathrm{ASM}_1, \mathrm{TLGF}_1)$ via a reduction to $\mathrm{FinSat}(\mathrm{TLGF}_1, (\mathbb{N}, <))$.
Let $\Pi \in \mathrm{ASM}_1$ and φ be a TLGF_1-formula over the same vocabulary as Π. We construct a formula Φ_Π such that

$$\begin{array}{c}(\Pi, \varphi) \in \mathrm{GVerify}(\mathrm{ASM}_1, \mathrm{TLGF}_1) \text{ if, and only if,} \\ \mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi) \notin \mathrm{FinSat}(\mathrm{TLGF}_1, (\mathbb{N}, <))\end{array}$$

where trans_Π transforms a TLGF_1-formula into another TLGF_1-formula with respect to Π as follows. $\mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi) \in \mathrm{FinSat}(\mathrm{TLGF}_1, (\mathbb{N}, <))$ if, and only if, there exists a model $\mathcal{M} = ((\mathbb{N}, <), D, I)$ of $\Phi_\Pi \wedge \neg\varphi$ such that for all quasi-atoms $\alpha(\overline{x})$ not unifiable with the left-hand side of an update rule in Π, $\alpha(I(n)) = \alpha(I(m))$ for all $m, n \in \mathbb{N}$.

Note that this does *not* imply that for every model $\mathcal{M} = ((\mathbb{N}, <), D, I)$ of $\mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi)$ and for all quasi-atoms $\alpha(\overline{x})$ not unifiable with the left-hand side of an update rule in Π, $\alpha(I(n)) = \alpha(I(m))$ for all $m, n \in \mathbb{N}$.
The formula $\Phi_\Pi \in \mathrm{TLGF}_1$ is defined as follows

$$\Phi_\Pi = \mathrm{G}((\mathrm{consistent}_\Pi \rightarrow (\psi_\Pi \wedge \chi_\Pi)) \wedge ((\neg\mathrm{consistent}_\Pi) \rightarrow \mathrm{noChanges}_\Pi))$$

where $\mathrm{consistent}_\Pi$ is the GF-sentence from lemma 6.10, ψ_Π and χ_Π are the sentences from lemma 6.11 and $\mathrm{noChanges}_\Pi$ is the one from lemma 6.12.

Because $(\text{consistent}_\Pi \rightarrow (\psi_\Pi \wedge \chi_\Pi)) \wedge ((\neg\text{consistent}_\Pi) \rightarrow \text{noChanges}_\Pi)$ is a TLGF_1-formula without free variables, Φ_Π is a TLGF_1-formula.

The preceding construction prevents only changes of (quasi-)atoms unifiable with the left-hand side of an update-rule. The problem is that this does not ensure that the content of locations not concerned by the updates of the ASM might change. The formula Φ_Π does not prevent this.

It is not possible to control these locations via an additional formula connected conjunctively to the formula $\Phi_\Pi \wedge \neg\varphi$. The problem is that the arity of the relations is not bounded and therefore, one would have to quantify into temporal contexts with more than one variable (e.g. $\forall\overline{x}(\text{G}R\overline{x} \vee \text{G}\neg R\overline{x})$) for a relation R which is never updated (therefore, it is static). A further problem is that this formula is not in the guarded fragment.

But there is another way for treating the problem that is less direct than the preceding approach. First, construct the TLGF_1-formula $\Phi_\Pi \wedge \neg\varphi$ as described above. Then transform it into another TLGF_1-formula where the idea for the transformation is the following.

The left-hand side of an update rule contains at most one variable. Therefore, it is of the form $R\overline{c}_1 x \overline{c}_2 ... \overline{c}_i x \overline{c}_{i+1} ... \overline{c}_m$ where $\overline{c}_i, i \in \{1, ..., m\}$ are (possibly empty) tuples of constant symbols. Let $i_1, ..., i_n$ be those positions in the tuple $\overline{b} = b_1 ... b_{\text{arity}(R)} = \overline{c}_1 x \overline{c}_2 ... \overline{c}_i x \overline{c}_{i+1} ... \overline{c}_m$ where x appears (i.e., for all $j \in \{1, ..., n\}$ $b_{i_j} = x$ and therefore $\{1, ..., \text{arity}(R)\} - \{i_1, ..., i_n\}$ are those where constant symbols appear). Consequently, the update rule does not concern those locations $(R, \overline{a}), \overline{a} = a_1 ... a_{\text{arity}(R)}$ where one of the following holds

1. there exist $l, k \in \{i_1, ..., i_n\}$ such that $a_k \neq a_l$ or
2. there exists $l \in \{1, ..., \text{arity}(R)\} - \{i_1, ..., i_n\}$ such that $a_i \neq b_i$

If there is more than one update rule for a relation R then the same is true for the other update rules.

Now, we give the formal description of the transformation trans_Π. It is defined in a number of steps.

For explaining the idea of the first step, consider the case of only one update rule for a relation R. The update set induced by this rule only concerns those locations $(R, \overline{a}), \overline{a} = a_1 ... a_{\text{arity}(R)}$ where

1. $a_k = a_l$ for all $l, k \in \{i_1, ..., i_n\}$ and
2. $a_i = b_i$ for all $l \in \{1, ..., \text{arity}(R)\} - \{i_1, ..., i_n\}$

(This is simply the negation of the above condition.)

Therefore, all other locations with this relation symbol can be considered as static. Those locations have the content true in a later state if, and only if, they have the content true in the first state. Note that the interpretation of a

constant symbol is not changed in the run of an ASM in ASM_1 and concerning satisfiability, we take into account only those models of a TLGF_1-formula where the interpretation of constant symbols is static. Therefore, any atomic statement about the relation R can be divided into two statements:

1. one for dynamic locations (possibly concerned by the update) and
2. one for the static locations

For the first part, we have to consider the flow of time because here, the changes indicated by Π happen.

The interpretation of the second part depends on the interpretation of R in the first state.

This argumentation can be directly transfered to more than one update rule for a relation symbol by considering all update rules for one relation symbol.

In the following, we give the transformation tr_1^Π. The description of tr_1^Π makes use of some definitions that we give in advance.

Let α be an atomic formula and R be the relation symbol of α. (Therefore, α has the form $R\overline{u}$ where $\overline{u}$ is tuple consisting of variables and constant symbols.) Let U_α^Π be the set of all atoms occurring on the left-hand side of an update rule in Π and where R is the relation symbol.

$$\alpha_\Pi := \bigwedge_{R\overline{z}\in U_\alpha^\Pi} \neg\Big(\bigwedge_{i\in\{1,\ldots,n\},\ z_i \text{ is a constant}} z_i = y_i \wedge \bigwedge_{i,j\in\{1,\ldots,n\},\ i\neq j,\ z_i \text{ is } x,\ z_j \text{ is } x} y_i = y_j\Big)$$

Obviously, $\mathrm{free}(\alpha) = \mathrm{free}(\alpha_\Pi)$.

The formula α_Π says for an assignment $\sigma : \mathrm{free}(\alpha) \to \mathrm{domain}(\mathcal{A})$ that $\sigma(\alpha)$ is not unifiable with an element from U_α^Π resp. $\sigma(\alpha)$ is not an instance of an element in U_α^Π. Another explanation is the following. For a location with relation symbol R, α_Π formalizes exactly the conditions for a location to be static.

For an atom γ, let $\mathrm{free}^*(\gamma)$ be the set of all free variables and constants occurring in γ.

For a TLGF_1-formula ψ, $\mathrm{tr}_1^\Pi(\psi)$ is defined by induction on ψ.

1. if ψ is an atomic formula α then $\mathrm{tr}_1^\Pi(\psi)$ is the following formula

$$(\alpha_\Pi \to \alpha) \wedge \bigwedge_{\beta\in U_\alpha^\Pi, \sigma_{\alpha,\beta}:\mathrm{free}^*(\alpha)\to\mathrm{free}^*(\beta)} ((\bigwedge_{z\in\mathrm{free}^*(\alpha)} z = \sigma_{\alpha,\beta}(z)) \to \sigma_{\alpha,\beta}(\alpha))$$

$$\equiv (\alpha_\Pi \to \alpha) \wedge \bigwedge_{\beta\in U_\alpha^\Pi, \sigma_{\alpha,\beta}:\mathrm{free}^*(\alpha)\to\mathrm{free}^*(\beta)} \sigma_{\alpha,\beta}(\alpha)$$

Note that for any dynamic (quasi-)atom α, $|\mathrm{free}(\alpha_\Pi)| = |\mathrm{free}(\alpha)| \leq 1$ holds.

2. $\mathrm{tr}_1^\Pi(\neg\chi) = \neg\mathrm{tr}_1^\Pi(\chi)$
3. $\mathrm{tr}_1^\Pi(\chi_1 \wedge \chi_2) = \mathrm{tr}_1^\Pi(\chi_1) \wedge \mathrm{tr}_1^\Pi(\chi_2)$
4. $\mathrm{tr}_1^\Pi(\chi_1 \vee \chi_2) = \mathrm{tr}_1^\Pi(\chi_1) \vee \mathrm{tr}_1^\Pi(\chi_2)$
5. $\mathrm{tr}_1^\Pi(\exists\overline{x}\ \chi) = \exists\overline{x}\mathrm{tr}_1^\Pi(\chi)$
6. $\mathrm{tr}_1^\Pi(\forall\overline{x}\ \chi) = \forall\overline{x}\mathrm{tr}_1^\Pi(\chi)$
7. $\mathrm{tr}_1^\Pi(\chi_1\ \mathrm{U}\ \chi_2) = \mathrm{tr}_1^\Pi(\chi_1)\ \mathrm{U}\ \mathrm{tr}_1^\Pi(\chi_2)$
8. $\mathrm{tr}_1^\Pi(\chi_1\ \mathrm{S}\ \chi_2) = \mathrm{tr}_1^\Pi(\chi_1)\ \mathrm{S}\ \mathrm{tr}_1^\Pi(\chi_2)$

The effect of applying tr_1^Π is that we have now divided all atomic formulae into their dynamic part and their static part (respectively defined by quasi-atoms). Note that for a TLGF_1-formula ψ, $\mathrm{tr}_1(\psi)$ is not necessarily in TLGF though it is equivalent to a TLGF-formula (namely, ψ or by splitting the quantifications if the guards are splitted).

Now, we define the transformation tr_2^Π. For a TLGF_1-formula ψ, $\mathrm{tr}_2^\Pi(\psi)$ is defined by induction on ψ.

1. For a dynamic atom α, $\mathrm{tr}_2^\Pi(\alpha) = \mathrm{X}\ \alpha$.
   (Note that $\mathrm{X}\ \alpha \in \mathrm{TLGF}_1$ for a dynamic atom α because $|\mathrm{free}(\alpha)| \leq 1$).
2. For a static quasi-atom α, $\mathrm{tr}_2^\Pi(\alpha) = \alpha$.
3. $\mathrm{tr}_2^\Pi(\neg\chi) = \neg\mathrm{tr}_2^\Pi(\chi)$
4. $\mathrm{tr}_2^\Pi(\chi_1 \wedge \chi_2) = \mathrm{tr}_2^\Pi(\chi_1) \wedge \mathrm{tr}_2^\Pi(\chi_2)$
5. $\mathrm{tr}_2^\Pi(\chi_1 \vee \chi_2) = \mathrm{tr}_2^\Pi(\chi_1) \vee \mathrm{tr}_2^\Pi(\chi_2)$
6. $\mathrm{tr}_2^\Pi(\exists\overline{x}\ \chi) = \exists\overline{x}\ \mathrm{tr}_2^\Pi(\chi)$
7. $\mathrm{tr}_2^\Pi(\forall\overline{x}\ \chi) = \forall\overline{x}\ \mathrm{tr}_2^\Pi(\chi)$
8. $\mathrm{tr}_2^\Pi(\chi_1\ \mathrm{U}\ \chi_2) = \mathrm{tr}_2^\Pi(\chi_1)\ \mathrm{U}\ \mathrm{tr}_2^\Pi(\chi_2)$
9. $\mathrm{tr}_2^\Pi(\chi_1\ \mathrm{S}\ \chi_2) = \mathrm{tr}_2^\Pi(\chi_1)\ \mathrm{S}\ \mathrm{tr}_2^\Pi(\chi_2)$

Essentially, tr_2^Π puts an X in front of every dynamic atom (not appearing inside a static quasi-atom). As for every dynamic atom α, $|\mathrm{free}(\alpha)| \leq 1$ holds, $\mathrm{tr}_2^\Pi(\psi)$ is contained in TLGF_1 for every TLGF_1-formula ψ.

Furthermore, we map every TLGF_1-formula ψ to a set of formulae $\mathrm{set}_1^\Pi(\psi)$. Let Q_ψ^Π be the set of static quasi-atoms appearing in ψ (with respect to Π).

In order to shorten the definition of $\mathrm{set}_1^\Pi(\psi)$, we define the sets G_ψ^Π and S_ψ^Π in advance.

Let G_ψ^Π be the set of all GF-formulae that are built from the elements of Q_ψ^Π. I.e., G_ψ^Π is the minimal set satisfying the following conditions.

1. $Q_\psi^\Pi \subseteq G_\psi^\Pi$
2. if $\chi_1 \in G_\psi^\Pi$ and $\chi_2 \in G_\psi^\Pi$ then
   - $\neg\chi_1 \in G_\psi^\Pi$
   - $\chi_1 \wedge \chi_2 \in G_\psi^\Pi$
   - $\chi_1 \vee \chi_2 \in G_\psi^\Pi$
3. if $\overline{x}, \overline{y}$ are tuples of variables, $\alpha(\overline{x}, \overline{y}) \in Q_\psi^\Pi$, $\chi(\overline{x}, \overline{y}) \in G_\psi^\Pi$, and $\mathrm{free}(\chi) \subseteq \mathrm{free}(\alpha) = \{\overline{x}, \overline{y}\}$ then
   - $\exists\overline{y}(\alpha(\overline{x}, \overline{y}) \wedge \chi(\overline{x}, \overline{y})) \in G_\psi^\Pi$
   - $\forall\overline{y}(\alpha(\overline{x}, \overline{y}) \rightarrow \chi(\overline{x}, \overline{y})) \in G_\psi^\Pi$

Note that if $\mathrm{free}(\chi) \subseteq \mathrm{free}(\alpha)$, $\alpha = \beta \wedge \xi \in Q_\psi^\Pi$ and β is an atomic formula, then $\forall\overline{x}(\alpha \rightarrow \chi)$, is equivalent to a formula in GF as $(\beta \wedge \xi) \rightarrow \chi \equiv \beta \rightarrow (\xi \rightarrow \chi)$. Therefore, the formalization does not contradict to the original (less formal) definition.

S_ψ^Π is the subset of G_ψ^Π containing all formulae $\chi \in G_\psi^\Pi$ with length $|\chi| \leq 2|\psi|$ and $|\mathrm{free}(\chi)| \leq 1$.

Using the above, we formulate the following definition:

$$\mathrm{set}_1^\Pi(\psi) := \{\forall x(\mathrm{G}\chi \vee \mathrm{G}\neg\chi) : \chi \in S_\psi^\Pi, \mathrm{free}(\chi) = \{x\}\}$$

The formulae in $\mathrm{set}_1^\Pi(\psi)$ ensure that for every element a of the domain the set

$$\{\chi \in S_\psi^\Pi : \mathrm{free}(\chi) \subseteq \{x\}, \mathcal{A} \models \chi[x/a]\}$$

is static (where $\mathcal{A}$ is any state of the considered temporal structure). This set of formulae can be considered as the type of a restricted to static quasi-atoms and guarded formulae of length $\leq 2 \cdot |\psi|$. Therefore, any formula possibly of interest for the valuation of ψ and which is built *only* from static atoms never changes its valuation.

Furthermore, we map every formula ψ to sets of formulae $\mathrm{set}_2^\Pi(\psi)$ and $\mathrm{set}_3^\Pi(\psi)$ is defined by induction on ψ where Υ_c is again the set of all constant symbols of Υ. The definition of set_2^Π involves the mapping set_3^Π and the definition of set_3^Π involves the mapping set_2^Π. In spite of this entangled definitions, both are well-founded.

1. For a dynamic atom α or a static quasi-atom α with $|\text{free}(\alpha)| \geq 2$, $\text{set}_2^\Pi(\alpha) = \emptyset$.

2. For a static quasi-atom α with $|\text{free}(\alpha)| \leq 1$, $\text{set}_2^\Pi(\alpha) = \forall x(\text{G}\alpha \vee \text{G}\neg\alpha)$ where $\text{free}(\alpha) \subseteq \{x\}$.

3. $\text{set}_2^\Pi(\neg\chi) = \text{set}_2^\Pi(\chi)$

4. $\text{set}_2^\Pi(\chi_1 \wedge \chi_2) = \text{set}_3^\Pi(\chi_1) \cup \text{set}_3^\Pi(\chi_2)$

5. $\text{set}_2^\Pi(\chi_1 \vee \chi_2) = \text{set}_3^\Pi(\chi_1) \cup \text{set}_3^\Pi(\chi_2)$

6. $\text{set}_2^\Pi(\exists\overline{x}(\alpha(\overline{x}) \wedge \chi)) = \text{set}_3^\Pi(\alpha(\overline{x})) \cup \text{set}_3^\Pi(\chi)$

7. $\text{set}_2^\Pi(\forall\overline{x}(\alpha(\overline{x}) \rightarrow \chi)) = \text{set}_3^\Pi(\alpha(\overline{x})) \cup \text{set}_3^\Pi(\chi)$

8. $\text{set}_2^\Pi(\chi_1 \text{ U } \chi_2) = \text{set}_3^\Pi(\chi_1) \cup \text{set}_3^\Pi(\chi_2)$.

9. $\text{set}_2^\Pi(\chi_1 \text{ S } \chi_2) = \text{set}_3^\Pi(\chi_1) \cup \text{set}_3^\Pi(\chi_2)$

For a formula $\psi \in \text{TLGF}_1$, $\text{set}_3^\Pi(\psi)$ is defined as follows. If $|\text{free}(\psi)| \leq 1$ (w.l.o.g., $\text{free}(\psi) = \{x\}$) then

$$\text{set}_3^\Pi(\psi) = \text{set}_2^\Pi(\psi) \cup \{\forall x(\text{G}((X\psi) \leftrightarrow \text{tr}_2^\Pi(\psi)))\}$$

else

$$\text{set}_3^\Pi(\psi) = \text{set}_2^\Pi(\psi)$$

The formulae in $\text{set}_3^\Pi(\psi)$ ensure that the interpretation of a TLGF_1-formula in the successor state depends on the interpretation of the static quasi-atoms in the current one and the changed interpretations of the dynamic atoms.

Now, we are ready to define for a ASM $\Pi \in \text{ASM}_1$ and a TLGF_1-formula ψ the formula $\text{trans}_\Pi(\psi)$.

$$\text{trans}_\Pi(\psi) = \psi \wedge \Big(\bigwedge_{\chi \in \text{set}_1^\Pi(\text{tr}_1^\Pi(\psi))} \chi\Big) \wedge \Big(\bigwedge_{\chi \in \text{set}_3^\Pi(\text{tr}_1^\Pi(\psi))} \chi\Big)$$

As $\text{tr}_1^\Pi(\psi))$ and ψ are equivalent, one might think about taking the conjunction of the formulae in $\text{set}_1^\Pi(\psi)$ resp. $\text{set}_3^\Pi(\psi)$ instead of the conjunction of the formulae in $\text{set}_1^\Pi(\text{tr}_1^\Pi(\psi))$ resp. $\text{set}_3^\Pi(\text{tr}_1^\Pi(\psi))$. The reason why we are not doing this is that, in ψ, the static and dynamic (quasi-)atoms are not separated resp. the atoms are not divided into their dynamic part and their static part. Consequently, the definitions of set_1^Π and set_3^Π can not handle the formula without applying tr_1^Π as atoms are usually not only dynamic or only static. Of course this could be eliminated by giving expanded versions of the definitions of set_1^Π and set_3^Π.

Subsequent to the definition of trans_Π, we prove that $\mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi)$ has the properties claimed at the beginning of the proof.

If there exists a finite model of Φ_Π such that for all quasi-atoms $\alpha(\overline{x})$ not unifiable with the left-hand side of an update rule in Π, $\alpha(I(n)) = \alpha(I(m))$ for all $m, n \in \mathbb{N}$ then there exists also a finite model of $\mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi)$ as the condition on the interpretation of static quasi-atoms ensures that $(\bigwedge_{\chi \in \mathrm{set}_1^\Pi(\mathrm{tr}_1^\Pi(\psi))} \chi) \wedge (\bigwedge_{\chi \in \mathrm{set}_3^\Pi(\mathrm{tr}_1^\Pi(\psi))} \chi)$ is satisfied.

Now, consider the other direction. Let $\mathcal{M}$ be a model of $\mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi)$. By definition, the following items hold.

1. $\mathcal{M} \models \Phi_\Pi \ \wedge \neg\varphi$
2. $\mathcal{M} \models \psi$ for all $\psi \in \mathrm{set}_1^\Pi(\mathrm{tr}_1^\Pi(\Phi_\Pi \ \wedge \neg\varphi))$
3. $\mathcal{M} \models \psi$ for all $\psi \in \mathrm{set}_3^\Pi(\mathrm{tr}_1^\Pi(\Phi_\Pi \ \wedge \neg\varphi))$

We now give a (finite) model of $\Phi_\Pi \ \wedge \neg\varphi$ such that for all quasi-atoms $\alpha(\overline{x})$ not unifiable with the left-hand side of an update rule in Π, $\alpha(I(n)) = \alpha(I(m))$ for all $m, n \in \mathbb{N}$.

Items 2. and 3. do not exclude that there exists a quasi-atom α which is not unifiable with the left-hand side of an update rule in Π (or to express it in different words, which is not concerned by an update rule in Π) but whose interpretation is changed in $\mathcal{M}$.

Let $\mathcal{M} = ((\mathbb{N}, <), D, I)$ and $\mathcal{M}^\Pi = ((\mathbb{N}, <), D, I')$ be the run of Π on $(D, I(1))$. I.e. $I'(1) = I(1)$ and $(D, I'(n)) \vdash_\Pi (D, I'(n+1))$ for all $n \in \mathbb{N}$. By construction, $\mathcal{M}_\Pi \models \Phi_\Pi \wedge (\bigwedge_{\chi \in \mathrm{set}_1^\Pi(\mathrm{tr}_1^\Pi(\Phi_\Pi \wedge \neg\varphi))} \chi) \wedge (\bigwedge_{\chi \in \mathrm{set}_3^\Pi(\mathrm{tr}_1^\Pi(\Phi_\Pi \wedge \neg\varphi))} \chi)$. In the remainder of the proof we show that $\mathcal{M}^\Pi \models \neg\varphi$.

In order to satisfy $\mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi)$, the interpretation of the static quasi-atoms is not allowed to change arbitrarily. At least, the constraints imposed by the formulae in $\mathrm{set}_3^\Pi(\mathrm{tr}_1^\Pi(\Phi_\Pi \wedge \neg\varphi))$ and $\mathrm{set}_1^\Pi(\mathrm{tr}_1^\Pi(\Phi_\Pi \wedge \neg\varphi))$ have to be satisfied.

The formulae in $\mathrm{set}_1^\Pi(\mathrm{tr}_1^\Pi(\Phi_\Pi \wedge \neg\varphi))$ ensure that, for each element a in the domain D, the set of formulae

$$\{\chi \in S_\psi^\Pi : \mathrm{free}(\chi) \subseteq \{x\}, \mathcal{A} \models \chi[x/a]\}$$

is the same for every state $\mathcal{A}$ in $\mathcal{M}$. To put it in different words, the type of an element a restricted to static quasi-atoms and guarded formulae of length $\leq 2|\psi|$ is static in $\mathcal{M}$.

In TLGF_1, it is not allowed to quantify into temporal contexts with more than one free variable. Therefore, the formulae in $\mathrm{set}_3^\Pi(\mathrm{tr}_1^\Pi(\Phi_\Pi \wedge \neg\varphi))$ ensure that, in a fixed state of $\mathcal{M}$, the interpretation of formulae whose temporal evaluation is considered (these are formulae ψ appearing in the form ρ U ψ, ψ U ρ, ρ S ψ or

ψ S ρ) does not depend on the interpretation of static subformulae (esp. quasi-atoms) in the current state but in its predecessor.

W.l.o.g., we can assume that the guard of an if-clause contains at most one free variable (compare normal form). Consider an existential formula $\psi = \exists\overline{x}(\alpha(\overline{x}) \wedge \varphi)$ where α is the guard of the formula. W.l.o.g., we can assume that α is static. The reason, why this assumption is admissible, is that dynamic atoms contain ≤ 1 variables. Therefore, if α is dynamic, ψ is equivalent to the TLGF_1-formula $\exists x(x = x \wedge (\alpha(\overline{x}) \wedge \varphi))$ where α is not the guard but $x = x$ which is static. Else, the following equivalences hold.

$$\exists\overline{x}(\alpha(\overline{x}) \wedge \psi)$$

$$\equiv \exists\overline{x}(\mathrm{tr}_1^{\Pi}(\alpha(\overline{x})) \wedge \psi)$$

$$\equiv \exists\overline{x}((\alpha_{\Pi}(\overline{x}) \rightarrow \alpha(\overline{x})) \wedge \bigwedge_{\beta \in U_{\alpha}^{\Pi}, \sigma_{\alpha,\beta}:\mathrm{free}^*(\alpha)\rightarrow\mathrm{free}^*(\beta)} (\bigwedge_{z \in \mathrm{free}^*(\alpha)} (z = \sigma_{\alpha,\beta}(z) \rightarrow \sigma_{\alpha,\beta}(\alpha)))) \wedge \psi)$$

$$\equiv (\exists\overline{x}(\alpha \wedge \alpha_{\Pi}(\overline{x}) \wedge \psi) \vee \bigvee_{\beta \in U_{\alpha}^{\Pi}, \sigma_{\alpha,\beta}:\mathrm{free}^*(\alpha)\rightarrow\mathrm{free}^*(\beta), \mathrm{free}(\beta)=\{y\}} \exists y(y = y \wedge \sigma_{\alpha,\beta}(\alpha) \wedge \sigma_{\alpha,\beta}(\psi))$$

Analogously to the argumentation in the existential case, we can show that we can assume, w.l.o.g., that the guard of a universal formula is static.

For a formula recommending in every state (or in a part of the states) the existence of elements satisfying a certain condition (given by the inner formula), it is not important whether the elements witnessing that the condition is satisfied are the same in every state or not. Therefore, instead of sometimes changing the witnessing elements, we can also take the same ones except that another part of the formula indicates changes except that other parts of the formula require changes. (Remember that it is not allowed to quantify *into* temporal contexts with more than one variable. For formulae with more than one free variable, there must be a quantifier inside the next outer temporal context.)

An argumentation similar to the previous one holds for universal quantifications.

If a subformula of $\mathrm{tr}_1^{\Pi}(\Phi_{\Pi} \wedge \neg\varphi)$ with at most one free variable is satisfied in a state $\mathcal{A}$ of $\mathcal{M}$ then it is also satisfied if $\mathcal{A}$ is exchanged with the state $\mathcal{A}'$ resulting from $\mathcal{A}$ by replacing the interpretation of static quasi-atoms by the interpretation of the static quasi-atoms in the predecessor state of $\mathcal{A}$ in $\mathcal{M}$. The change in the interpretation of the static quasi-atoms is limited in $\mathcal{M}$.

Until now, the argumentation is more or less local in the sense that we consider only the change of the interpretation for each single state and its predecessor. Global considerations are necessary in order to ensure that $\neg\varphi$ holds in $\mathcal{M}_\Pi$. The formulae in $\text{set}_1^\Pi(\text{tr}_1^\Pi(\Phi_\Pi \wedge \neg\varphi))$ allow to extend the above to the complete structure $\mathcal{M}$ resulting in the construction of $\mathcal{M}_\Pi$. Without the formulae in $\text{set}_1^\Pi(\text{tr}_1^\Pi(\Phi_\Pi \wedge \neg\varphi))$, it would even not be possible to extend the above considerations.

In the case of universal quantifications, the only reason is that we might empty a static relation in the model $\mathcal{M}$. Consider a formula of the form $\psi = \forall\overline{x}(R\overline{x} \rightarrow \rho)$ and two states $\mathcal{A}_1$, $\mathcal{A}_2$ where $\mathcal{A}_2$ is the successor of $\mathcal{A}_1$. Assume furthermore that there is at least one tuple $\overline{a}$ of elements from the domain of $\mathcal{A}_1$ satisfying α ($\{\overline{a} : \mathcal{A}_2 \models \alpha[\overline{x}/\overline{a}]\} \neq \emptyset$). According to the formulae in $\text{set}_3^\Pi(\text{tr}_1^\Pi(\psi))$, the interpretation/valuation of ψ in $\mathcal{A}_2$ depends on the interpretation of α in $\mathcal{A}_1$. But possibly, the set $\{\overline{a} : \mathcal{A}_2 \models \alpha[\overline{x}/\overline{a}]\}$ is empty. Therefore, the conditions defined by the formulae in $\text{set}_1^\Pi(\text{tr}_1^\Pi(\psi))$ are empty for the successor of $\mathcal{A}_2$. The conditions imposed by the formulae in $\text{set}_3^\Pi(\text{tr}_1^\Pi(\psi))$ exclude this possibility.

An argumentation similar to the previous one holds for existential quantifications. In this case, emptying a relation is no problem but the possibility of adding all tuples of elements from the domain to the relation is the corresponding problem. $\text{set}_1^\Pi(\text{tr}_1^\Pi(\Phi_\Pi \wedge \neg\varphi))$ does also prevent this possibility.

Therefore, $\mathcal{M}^\Pi$ is a model of $\Phi_\Pi \wedge \neg\varphi$ such that for all quasi-atoms $\alpha(\overline{x})$ not unifiable with the left-hand side of an update rule in Π, $\alpha(I(n)) = \alpha(I(m))$ for all $m, n \in \mathbb{N}$. □

6.4 Complexity

We first consider the complexity of $\text{FinSat}(\text{TLGF} \cap \mathcal{TL}_1, (\mathbb{N}, <))$ in order to determine an upper bound for the complexity of $\text{GVerify}(\text{ASM}_1, \text{TLGF}_1)$,

The proof in [24] resp. [25] implicitly provides an algorithm deciding the language $\text{FinSat}(\text{TLGF}_1, (\mathbb{N}, <))$. For a formula $\varphi \in \text{TLGF}_1$, the runtime of this algorithm is in

$$2^{2^{O(2^{|\varphi|} * \log|\varphi|)}}$$

Now, we analyze the complexity of $\text{GVerify}(\text{ASM}_1, \text{TLGF}_1)$. The size of the formula $\Phi_{\Pi,\varphi}$ is linear in the length of (Π, φ) (we refer to it as $|(\Pi, \varphi)|$). The transformation blows up the size of the formula. In the worst case, each step of the transformation doubles the size of the formula. The number of steps is bounded number of tuples of constants which are relevant for the transformation. Therefore, an upper bound for the size of $(\tilde{\Phi}_{\Pi,\varphi})$ is $2^{\max\{arity(R):R\in\Upsilon\}} \cdot |\Phi_{\Pi,\varphi}|$.

Therefore, the time complexity of $\text{GVerify}(\text{ASM}_1, \text{TLGF}_1)$ is in

$$2^{2^{O(2^{2^{\max\{arity(R):R\in\Upsilon\}}}\cdot|(\Pi,\varphi)|*(\max\{arity(R):R\in\Upsilon\}+\log|(\Pi,\varphi)|)})}.$$

For the following fragments, we obtain a smaller lower bound:

Bounded arity. In this case, the factor $2^{\max\{arity(R):R\in\Upsilon\}}$ can be replaced by a constant factor.

Bounded number of constant symbols. In this case, we can replace $\log|(\Pi,\varphi)|$ by a constant.

In the case of nondeterministic time, we have one exponent less.

Because of the lower bounds proven in [13] (and mentioned above), double-exponential time is a lower bound for the time complexity of the considered verification problem (and exponential time is a lower bound in the case of bounded arities).

6.5 Extension by Interaction

The main difference between ASM transducers (see [32] or section 5.2) and ASMs in ASM_1 is that ASM transducers provide the possibility to interact with their environment. The remaining differences, as the static database part, the output part or the memory part of the transducers, are not really additional features. They can be immediately integrated into the framework of ASM_1.

ASM_1 can be extended by interaction in the following way. Instead of one input state (over the the vocabulary Υ of the program) in a run, there is a sequence of input states $\overline{I} = I_1 I_2 ...$ over the input vocabulary $\Upsilon_{in} \subseteq \Upsilon$. The notion of a run has to be modified as follows:
Let ρ be a sequence $(\mathcal{A}_i)_{i\in\mathbb{N}}$ of states over Υ. ρ is the run of Π on $\overline{I}$ if

- $\mathcal{A}_i|_{\Upsilon_{in}} = I_i$
- Let $\mathcal{B}_{i+1}$ be the successor of $\mathcal{A}_i$ with respect to Π. Then $\mathcal{A}_{i+1}|_{\Upsilon-\Upsilon_{in}} = \mathcal{B}_{i+1}|_{\Upsilon-\Upsilon_{in}}$ for all $i \in \mathbb{N}$.

We denote the class of interactive ASMs resulting from ASM_1 by ASM_1^i.

The decidability result for ASMs in ASM_1 can be directly transferred to ASMs in ASM_1^i. Note that in the case of interactive ASMs, the general verification problem has to be slightly modified. It does not suffice to consider just all input states. We have to take into account all input states and all sequences of states over the input vocabulary.

On the first view, one may have the impression that the results from [32] are stronger than those presented in this thesis. The reason for this impression

might be the restrictions on the use of variables in ASMs from ASM_1. But this impression is wrong.

The first and obvious point is that the verifiable properties are not the same. For ASM-transducers, we can verify properties expressible in UT, for interactive ASMs in ASM_1 we can verify properties expressible in $\mathrm{TCGF}_1 := \mathcal{TL}_1 \cap \mathrm{CGF(LTFO)}$. The two fragments do not coincide and for each of the two fragments there are properties expressible in the one but not in the other.

But this observation does not mirror the essential difference between the two classes of formalisms. We demonstrate that by an example. In this example, we do not consider an ASM from ASM_1^i but an ASM from a slight extension of ASM_1^i. The restriction is not the one derived from the guarded fragment but the one from the clique-guarded fragment (CGF) of first-order logic. We denote the corresponding class of interactive ASMs by ASM_1^{cgi}. The decidability of this instance of the general verification problem can be proven analogously to theorem 6.13. Using ASMs in ASM_1^{cgi} , we want to model a very simple system governing the student data of a university.

At the end of his studies, a student has to take part in the exams of at least r combinable lectures. Only if he passes all exams then he receives a certificate. If he fails then he has the possibility to do r combinable exams again. If this was already his second try then he has to leave the university without a certificate. After every term, the results of the exams are submitted to the system and the current certificates and aborts are emitted. Internally, students are identified via their registration number. This allocation is static. Output data like aborts or certificates does not contain registration numbers.

We now give an ASM-transducer and an interactive clique-guarded ASM which are equivalent and both try to model the above described system.

First, we present the ASM-transducer.

relations

input	:	fail/2, pass/2
database	:	number/2, combinable/r
memory	:	try1/1, try2/1
output	:	certificate/1, abort/1
log	:	fail/2, pass/2, certificate/1, abort/1

memory rules

if exam(n) **then** try1(n)

if exam(n) $\wedge$ try1(n) **then** try2(n)

output rules

if number$(s, n) \wedge (\exists l$ fail$(n, l)) \wedge$ try1(n) **then** abort(s)

if number$(s, n) \wedge \neg$try2$(n) \wedge \neg(\exists l$ fail$(n, l)) \wedge$
$(\exists l_1 ... l_r$ combinable$(l_1, ..., l_r) \wedge \bigwedge_{i \in \{1,...,r\}}$ pass$(n, l_i)) \wedge$ **then**
certificate(s)

From the informal description, the work of the transducer should be clear. Nevertheless, we will partially resume the intended meaning of the relations.

For a registration number n and a lecture l, pass(n, l) is true if the student with registration number n has passed the exam belonging to l in the current term. fail(n, l) is true if he has failed the exam. The binary relation number/2 stores the allocation of registration numbers and students. The memory relations keep track of the number of terms in which a student has taken part in the exams. try1(n) is true if the student belonging to number n has already tried once to pass the exams (albeit whether passed or failed), try2(n) is true if he has taken part in the exams of two terms.

The above ASM-transducer can be rewritten to an equivalent ASM in ASM_1^{cgi} with the same input vocabulary. To increase the readability, do-in-parallel is left away and the rules are split into memory rules and output rules.

memory rules

forall n : true **do**
  if exam(n) **then** try1(n) := true **endif**
  if exam$(n) \wedge$ try1(n) **then** try2(n) := true **endif**
endforall

output rules

forall s, n : number(s, n) **do**
  if try1$(n) \wedge \exists l$ fail(n, l) **then**
    abort(s) := true
  else abort(s) := false
  endif
  if $\neg$try2$(n) \wedge \neg(\exists l$ fail$(n, l)) \wedge$
    $(\exists l_1 ... l_r$combinable$(l_1, ..., l_r) \wedge$
      $\bigwedge_{i \in \{1,...,r\}}$ pass$(n, l_i))$ **then**
    certificate(s) := true
  else certificate(s) := false
  endif
endforall

forall s : true **do**
  if $\neg\exists n$ number(s, n) **then**
    certificate(s) := false
    abort(s) := false
  endif
endforall

where exam(n) abbreviates the formula $(\exists l$ pass$(n, l)) \vee (\exists l$ fail$(n, l))$.

With the results presented in this paper, we can verify any property expressible by a TCGF_1-formula without any further restrictions. The results in [32] do not yield this possibility for UT as we have to care for the restrictions introduced there. In our example, a fixed database would determine the number of students.

A limited input flow would imply a limit on the number of exams albeit whether passed or failed. Both restrictions are not suitable in the above case.

An example for a property expressible in TCGF_1 is that if someone fails for the second time then he will never receive a certificate in the future. It can be formalized by the following TCGF_1-formula.

$$\begin{aligned} &\mathrm{G}\ \forall n(\mathrm{failed}(n) \rightarrow \mathrm{X}\ \mathrm{G}\ \mathrm{try1}(n)) \quad \wedge \\ &\mathrm{G}\ \forall sn(\mathrm{number}(s,n) \rightarrow ((\mathrm{try1}(n) \wedge \mathrm{failed}(n)) \rightarrow \neg \mathrm{F}\ \mathrm{certificate(s)})) \end{aligned}$$

where $\mathrm{failed}(n)$ abbreviates the formula $\exists l\ \mathrm{fail}(n,l)$.

Note, furthermore, that, in contrast to ASM-transducers, the memory relations of an ASM in ASM_1^{cgi} do not need to be initially empty but if we want to verify a property φ only for this case then we can do it by verifying the formula $(\bigwedge_{R\in\Upsilon_{mem}} \neg\exists x_1 ... x_{\mathrm{arity}(R)} R x_1 ... x_{\mathrm{arity}(R)}) \rightarrow \varphi$ instead of φ (where Υ_{mem} is the set of memory-relations of the transducer).

Verification of the Extension by Interaction

To transfer the decidability result to the extension by interaction, we do not have to change many things. It is enough to adapt the proof in the following way.

The arbitrary changes of the input relations resp. the locations with a relation symbol from the input vocabulary, need not to be avoided. W.l.o.g., we can assume that the input relations do not appear on the left-hand side of an update rule (because their update would not have any influence on the next state). So, the only thing, where something has to be changed is the transformation. The input relations have to be considered as completely dynamic and therefore, we have to consider some additional cases for the input relations.

7 Monadic Abstract State Machines

In this chapter, we introduce a further new instance of the general verification problem that is decidable. This instance has been mentioned only briefly in [27].

7.1 Definition and a Normal Form

Definition 7.1. A monadic ASM is an ASM in $\mathcal{D}$ that satisfies the following conditions.

- the vocabulary contains only constant symbols and unary relation symbols
- import is not used
- the left-hand side of an update-rule is not a constant symbol

The class of all monadic ASMs is denoted by ASM^{mo}.

Again, this class of ASMs is stronger than a well-defined fragment of first-order fixed point logic.

This one is obtained from general first-order fixed point logic by only allowing only relation symbols that are at most unary. This restriction applies to both, the symbols in the vocabulary and the fix point variables. Guards are not needed anymore.

As for ASMs in ASM_1 (compare lemma 6.2), there exists also a normal form for monadic ASMs.

Lemma 7.2. For every monadic ASM, we can effectively construct an equivalent monadic ASM of the following form:

do-in-parallel
  Π_1
  .
  .
  .
  Π_n
enddo

where each of the programs Π_i, $i \in \{1, .., n\}$ is of the form

forall x : true **do**
  if $\varphi_t^{\beta(x)}$ **then** $\beta(x) :=$ true **endif**
endforall

or of the form

forall x : true **do**
  if $\varphi_f^{\beta(x)}$ **then** $\beta(x) :=$ false **endif**
endforall

where $\varphi_t^{\beta(x)}$ is equal to false if an update of $\beta(x)$ to true is not possible and $\varphi_f^{\beta(x)}$ is equal to false if an update of $\beta(x)$ to false is not possible.

Furthermore, for every atomic formula $\beta(x)$, there appears at most one program of the first form and at most one of the second form.

Proof. We prove the lemma by effectively constructing for an arbitrary monadic ASM an equivalent monadic ASM of the described form.

Monadic ASMs can be transformed into a block of programs which have the form of the following program Π:

forall $x, \overline{y} : \varphi(x, \overline{y})$ **do**
  if ψ **then** $\alpha(x) := \gamma(x, \overline{y})$ **endif**
endforall

where $\alpha(x)$ is an atomic formula with free$(\alpha) \subseteq \{x\}$, $\psi, \varphi \in \mathrm{FO}^{mo}$, free$(\varphi) \subseteq \{x, \overline{y}\}$ and free$(\psi) \subseteq \{x, \overline{y}\}$.

Note that $\alpha(x)$ has either the form r where r is a nullary relation symbol from the vocabulary of the ASM or the form Rx where R is a unary relation symbol from the vocabulary of the ASM.

This form results from an application of the following equivalence preserving rewriting rules.

original rule	**equivalent rule**

forall $\overline{x} : \varphi(\overline{x}, \overline{y})$ **do**
  do-in-parallel
  Π_1
  Π_2
  enddo
endforall
	do-in-parallel
	  forall $\overline{x} : \varphi(\overline{x}, \overline{y})$ **do**
	  Π_1
	  endforall
	  forall $\overline{x} : \varphi(\overline{x}, \overline{y})$ **do**
	  Π_2
	  endforall
	enddo

if φ **then**
  do-in-parallel
  Π_1
  Π_2
  enddo
endif
	do-in-parallel
	  if φ **then** Π_1 **endif**
	  if φ **then** Π_2 **endif**
	enddo

if φ **then**
  forall $\overline{x} : \psi(\overline{x}, \overline{y})$ **do**
  Π'
  endforall
endif
	forall $\overline{x} : \psi(\overline{x}, \overline{y}) \wedge \varphi$ **do**
	  Π'
	endforall

forall $\overline{x} : \varphi(\overline{x}, \overline{z})$ **do**
  forall $\overline{y} : \psi(\overline{y}, \overline{x}, \overline{z})$ **do**
  Π'
  endforall
endforall
	forall $\overline{x}, \overline{y} : \varphi(\overline{x}, \overline{y}) \wedge \psi(\overline{y}, \overline{x}, \overline{z})$ **do**
	  Π'
	endforall

if φ **then**
  if ψ **then** $\tilde{\Pi}$ **endif**
endif
	if $\varphi \wedge \psi$ **then** $\tilde{\Pi}$ **endif**

The resulting program Π is equivalent to:

forall x : true **do**
  if $\exists \overline{y}(\psi(x, \overline{y}) \wedge \varphi(x, \overline{y}) \wedge \gamma(x, \overline{y}))$ **then** $Rx :=$ true **endif**

endforall
forall x : true **do**
if $\exists \overline{y}(\psi(x,\overline{y}) \wedge \varphi(x,\overline{y}) \wedge \neg\gamma(x,\overline{y}))$ **then** $Rx :=$ false **endif**
endforall

Furthermore, the program

forall x : true **do**
if ψ_1 **then** $\beta(x) := v$ **endif**
endforall
forall x : true **do**
if ψ_2 **then** $\beta(x) := v$ **endif**
endforall

is equivalent to

forall x : true **do**
if $\psi_1 \vee \psi_2$ **then** $\beta(x) := v$ **endif**
endforall

($v \in \{\text{true}, \text{false}\}$).

This implies the proposition of the lemma. □

7.2 Logical Background

Definition 7.3. By $\mathcal{TL}^{mo}$ we denote the monadic fragment of $\mathcal{TL}_1$. I.e., the fragment of $\mathcal{TL}_1$ using only relation symbols of arity ≤ 1. The use of equality is not allowed.

The fragment $\mathcal{TL}^{mo}$ is the logic for describing temporal properties of monadic ASMs and for which the general verification problem will be proven to be decidable.

In [25], the decidability of the following problem has been proven.

Given a sentence $\varphi \in \mathcal{TL}^{mo}$. Is φ satisfiable in $(\mathbb{N}, <)$ over a finite domain?

Definition 7.4. Denote by $\text{FinSat}(\mathcal{TL}^{mo}, (\mathbb{N}, <))$ the set of all $\mathcal{TL}^{mo}$-sentences which are satisfiable in $(\mathbb{N}, <)$ over finite, not expanding domains.

Denote by FO^{mo} the monadic fragment of first-order logic.

7.3 Deciding the General Verification Problem

We can prove the decidability of GVerify(ASM^{mo}, $\mathcal{TL}^{mo}$) analogously to theorem 6.13. The proof for ASM^{mo} monadic ASMs becomes even easier compared to ASM$_1$.

We construct the formula as in the proof of theorem 6.13 but the transformation is substituted by the conjunctive connection with a formula saying that all other locations are not changed. This possibility has been mentioned in the proof of theorem 6.13 and we explained why this formula does not satisfy the conditions for $\mathcal{TL}_1$ in case of ASM$_1$. The reason is that there might be relation symbols of arity greater than one so that the conditions for $\mathcal{TL}_1$ might not be satisfied. But for monadic ASMs, this is not possible as the arity is bounded by one.

So, the reduction is rather similar but much easier. We obtain a formula in $\mathcal{TL}^{mo}$. The decidability result from [25] implies the decidability of the general verification problem.

Nevertheless, we present a proof for the decidability of the general verification problem for monadic ASMs in order to see the difference and the simplifications.

Lemma 7.5. For every monadic ASM Π, there exists an effectively constructable sentence consistent$_\Pi \in$ FO^{mo} such that for all states $\mathcal{A}$ of Π the following holds: $\mathcal{A} \models$ consistent$_\Pi$ if, and only if, the update set produced by Π on $\mathcal{A}$ is consistent.

Proof. The construction of consistent$_\Pi$ is straightforward and completely analogous to the one for ASMs in ASM$_1$. Therefore, we do not repeat it. □

Lemma 7.6. For every monadic ASM Π there exists an effectively constructable sentence $\Psi_\Pi \in \mathcal{TL}^{mo}$ such that for all states $\mathcal{A}, \mathcal{B}$ of Π and all temporal structures $\mathcal{M} = ((\mathbb{N}, <), D, I)$ with $I(1) = \mathcal{A}$, $I(2) = \mathcal{B}$ and the update set produced by Π on $\mathcal{A}$ is consistent, the following holds:

$$\mathcal{A} \vdash_\Pi \mathcal{B} \text{ if, and only if, } (\mathcal{M}, 1) \models \Psi_\Pi$$

Proof. The formula Ψ_Π is the conjunction of two formulae ψ_Π and χ_Π. I.e., $\Psi_\Pi = \psi_\Pi \wedge \chi_\Pi$.

First, we construct the formula ψ_Π. It formulates the changes induced by Π and is defined by induction on Π:

- If Π has the form $(s := t)$ then $\psi_\Pi := (\mathrm{X}s) \leftrightarrow t$.
- If Π has the form **if** g **then** Π' **endif** then $\psi_\Pi := g \rightarrow \psi_{\Pi'}$.
- If Π has the form **do-in-parallel** Π_0 Π_1 **enddo** then $\psi_\Pi := \psi_{\Pi_0} \wedge \psi_{\Pi_1}$.

- If Π has the form **forall** $\overline{x}$: $\alpha(\overline{x},\overline{y})$ **do** $\Pi'(\overline{x},\overline{y})$ **endforall** then $\psi_\Pi := \forall\overline{x}(\alpha(\overline{x},\overline{y}) \rightarrow \psi_{\Pi'}(\overline{x},\overline{y}))$.

ψ_Π is a sentence because Π has no free variables. In ψ_Π, the only temporal operators are those X which are introduced via an update rule. Therefore, subformulae of ψ_Π with a temporal operator outermost are of the form Xs where s is an atomic formula. As all relation symbols are at most unary, s contains at most one free variable. Therefore $\psi_\Pi \in \mathcal{TL}^{mo}$.

χ_Π avoids changes in those atoms appearing on the left-hand side of an update-rule in Π not indicated by the program.

Assuming, w.l.o.g., that the program Π is in the normal form from lemma 7.2, we now construct the formula χ_Π.

For an arbitrary element a of the domain of the current state, the intended formula χ_Π says the following. If there are no assignments to $\overline{y}$ such that φ holds (with a assigned to x), then the interpretation of the atomic formula $\beta(a)$ does not change.

The formula is the conjunction of two formula, each treating one of the following two cases:

1. $\beta(x)$ (resp. with a assigned to x) is interpreted as true in the current state.
2. $\beta(x)$ (resp. with a assigned to x) is interpreted as false in the current state.

The first case is treated by the formula

$$\forall x(\beta(x) \rightarrow (\neg\varphi_f \rightarrow (X\beta(x)))).$$

The second case is treated by the formula

$$\forall x((\neg\beta(x)) \rightarrow (\neg\varphi_t \rightarrow (X\neg\beta(x)))).$$

Both formulae are in $\mathcal{TL}^{mo}$. □

If the update set produced by a program on a structure is not consistent then nothing changes.

Lemma 7.7. For every ASM $\Pi \in \mathrm{ASM}_1$ there exists an effectively constructable sentence $\mathrm{noChanges}_\Pi \in \mathcal{TL}^{mo}$ such that for all states $\mathcal{A},\mathcal{B}$ of Π and for all temporal structures $\mathcal{M} = ((\mathbb{N},<),D,I)$ with $I(1) = \mathcal{A}$ and $I(2) = \mathcal{B}$, the following holds.

$$(\mathcal{M},1) \models \mathrm{noChanges}_\Pi \text{ if, and only if, } \mathcal{B} \text{ is identical with } \mathcal{A}$$

Proof. Let $\Pi \in \mathrm{ASM}^{mo}$, Υ be the vocabulary of Π, $\Upsilon^0_{rel} \subseteq \Upsilon$ contain all nullary relation symbols of Υ and $\Upsilon^1_{rel} \subseteq \Upsilon$ contain all unary relation symbols of Υ. Then

$$\mathrm{noChanges}_\Pi = \bigwedge_{r\in\Upsilon^0_{rel}}(r \leftrightarrow \mathrm{X}\ r) \wedge \forall x \bigwedge_{R\in\Upsilon^1_{rel}}(Rx \leftrightarrow \mathrm{X}\ Rx)$$

□

Theorem 7.8. GVerify($\text{ASM}^{mo}, \mathcal{TL}^{mo}$) is decidable.

Proof. We prove the decidability of GVerify($\text{ASM}^{mo}, \mathcal{TL}^{mo}$) via a reduction to FinSat($\mathcal{TL}^{mo}, (\mathbb{N}, <)$).
Let Π be a monadic ASM program and φ be a $\mathcal{TL}^{mo}$-formula over the same vocabulary as Π. We construct a formula Φ_Π such that

$$(\Pi, \varphi) \in \text{GVerify}(\text{ASM}_1, \text{TLGF}_1) \text{ if, and only if,}$$
$$\Phi_\Pi \wedge \neg\varphi \notin \text{FinSat}(\mathcal{TL}^{mo}, (\mathbb{N}, <))$$

The formula $\Phi_\Pi \in \mathcal{TL}^{mo}$ can be defined as follows

$$\Phi_\Pi = \text{G}((\text{consistent}_\Pi \rightarrow \Psi_\Pi) \wedge ((\neg\text{consistent}_\Pi) \rightarrow \text{noChanges}_\Pi))$$

where consistent_Π is the GF-sentence from lemma 7.5, Ψ_Π is the sentence from lemma 7.6 and noChanges_Π is the one from lemma 7.7.
Because $(\text{consistent}_\Pi \rightarrow (\psi_\Pi \wedge \chi_\Pi)) \wedge ((\neg\text{consistent}_\Pi) \rightarrow \text{noChanges}_\Pi)$ is a $\mathcal{TL}^{mo}$-sentence, Φ_Π is a $\mathcal{TL}^{mo}$-sentence. □

7.4 Extension by Interaction

Analogously to ASMs in ASM_1, we can extend monadic ASMs by interaction. By ASM_i^{mo} we denote the class of interactive monadic ASMs.

Furthermore, we can easily prove the following complexity result.

Lemma 7.9. GVerify($\text{ASM}_i^{mo}, \mathcal{TL}^{mo}$) is EXPSPACE-complete.

Proof. The proof proceeds in the usual two steps. First, we prove that the language GVerify($\text{ASM}_i^{mo}, \mathcal{TL}^{mo}$) is in EXPSPACE. Then we give a reduction of a problem that is known to be EXPSPACE-complete. In this case, we choose FinSat($\mathcal{TL}^{mo}, (\mathbb{N}, <)$).

1. We know that FinSat($\mathcal{TL}^{mo}, (\mathbb{N}, <)$) is in EXPSPACE. Furthermore, the proof of theorem 7.8 provides a linear time reduction to FinSat($\mathcal{TL}^{mo}, (\mathbb{N}, <)$). Therefore, GVerify($\text{ASM}_i^{mo}, \mathcal{TL}^{mo}$) is in EXPSPACE.

2. In order to prove EXPSPACE-hardness, we reduce FinSat($\mathcal{TL}^{mo}, (\mathbb{N}, <)$) to GVerify($\text{ASM}_i^{mo}, \mathcal{TL}^{mo}$).

   Let $\varphi \in \mathcal{TL}^{mo}$ be an arbitrary formula (over the vocabulary Υ). Then $\varphi \in \text{FinSat}(\mathcal{TL}^{mo}, (\mathbb{N}, <))$ if, and only if, $(\textbf{Skip}, \neg\varphi) \notin \text{GVerify}(\text{ASM}_i^{mo}, \mathcal{TL}^{mo})$. In the context of *i*nteractive monadic ASMs, **Skip** is the program not changing the content of any location. But the locations with relations from the input vocabulary may change arbitrarily. In this context, we define the input vocabulary to contain all relations appearing in φ.

□

8 Further Verifiable Classes and Conclusion

8.1 Further Verifiable Classes of Abstract State Machines

In [25] and [24], the decidability of satisfiability problem over finite domains, $(\mathbb{N}, <)$ has been been proven for a number of fragments of $\mathcal{TL}_1$. For example, it has been proven for the clique-guarded fragment and the two-variable fragment.

In this chapter, we consider each of these fragments briefly. For each fragment, we present a class of ASMs for which the respective fragment (or to be more precise: the decidability of the satisfiability in $(\mathbb{N}, <)$ over finite domains for this fragment) implies the decidability of the general verification problem. As we have already seen two decidability proofs and the proofs of the following results are completely analogous to one of the two given proofs, we omit the proofs of the following three results.

8.1.1 The Clique-Guarded Fragment

In definition 1.5, the clique-guarded fragment of first-order logic has been introduced. It is an extension of the guarded fragment.

Analogously, we can extend the guarded fragment of a class $\mathcal{C}$ of ASMs to the clique-guarded fragment of $\mathcal{C}$.

Definition 8.1. The clique-guarded fragment $\mathrm{CGF}(\mathcal{C})$ of a class $\mathcal{C}$ of ASMs is the subset of $\mathcal{C}$ such that every element Π of $\mathrm{CGF}(\mathcal{C})$ satisfies the following conditions:

- the vocabularies contain only relation and constant symbols
- every guard of an if-clause in Π is in the clique-guarded fragment of first-order logic
- every forall-rule has the form **forall** $\overline{x} : \varphi(\overline{x}, \overline{y})R$ **endforall** where $\varphi(\overline{x}, \overline{y})$ is a clique-guard and $\mathrm{free}(R) \subseteq \mathrm{free}(\varphi) = \{\overline{x}, \overline{y}\}$

- every choice-rule (choice of an element from the domain) has the form **choose** $\overline{x} : \varphi(\overline{x}, \overline{y})R$ **endchoose** where $\varphi(\overline{x}, \overline{y})$ is an atomic formula and $\text{free}(R) \subseteq \text{free}(\varphi) = \{\overline{x}, \overline{y}\}$

The class of clique ASMs ASM_1^c is the well-defined subset of $\text{CGF}(\mathcal{D})$ containing exactly those ASMs in $\text{GF}(\mathcal{D})$ whose update rules have the form $R\overline{c}_1 x \overline{c}_2 x ... \overline{c}_n := t$.

The proof of the decidability of the verification problem for clique ASMs and formulae in the clique-guarded fragment of first-order temporal logic intersected with $\mathcal{TL}_1$ is the rather same as the one given for ASMs in ASM_1 and the guarded fragment.

The corresponding fixed point logic is obtained from the original one by again relaxing the condition on the guards.

8.1.2 The Two-Variable Fragment

In [25], the decidability of the satisfiability in $(\mathbb{N}, <)$ over finite domains has been proven for formulae in the two-variable fragment of $\mathcal{TL}_1$.

Leaving away all restrictions (except the restriction on the number of variables on the left-hand side of an update) made for ASMs in ASM_1 (as in the case of monadic formulae) and adding the restriction that at most two variables are allowed, leads to a further class of ASMs.

The decidability of the general verification problem for two-variable ASMs and properties specified in the two-variable fragment of $\mathcal{TL}_1$ can be proven via a reduction to the satisfiability problem of the two-variable fragment of $\mathcal{TL}_1$ in $(\mathbb{N}, <)$ over finite domains.

8.2 Conclusion

We have observed that the decidability results for both versions of the verification problem are tightly connected to decidability results in mathematical logic. This is the main reason for putting relatively strong restrictions on the ASMs and the specification logic in order to obtain positive decidability results.

As the worst-case complexities of the satisfiability problem for expressive logics are usually quite high (if decidable), the worst-case complexities for both versions of the verification problem are also quite high.

In order to obtain lower worst-case complexities, one has to accept a trade-off concerning the expressiveness. But in this case, the question for applicability becomes more and more difficult to answer. But we should also note that these high complexities reflect only the worst-case. For applications, we would be much more interested in the average-case complexity. It depends strongly on the particular applications.

This is essentially what we should keep in mind when trying to identify further verifiable fragments.

Part III

Slicing Abstract State Machines

9 Introduction to Slicing

9.1 Motivation of Slicing

The main observation from the preceding part is the existence of classes of ASMs and properties allowing automatic verification. At the same time, it is also clear that automatic verifiability has certain limits without any possibility to exceed them. Thus, the claim of verifying properties of ASMs automatically is desirable but too strong in some cases. Nevertheless, it is necessary to ensure that an ASM satisfies certain (required) properties.

In the past, a number of approaches has been suggested in order to close this gap. One example are translations to the formalisms of interactive provers as PVS. A further example is testing which can applied as ASMs are executable. Apart from this, the approach of testing has also been investigated in another direction. Test cases are generated from an ASM specification in order to check an implementation against its ASM specification.

In this part of the thesis, we present the concept of slicing ASMs intending to make the above techniques more precise. The idea is similar to the one of program slicing (see e.g. [34], [37]).

Usually, writing formal specifications is not an error-free process. Therefore, we are inevitably confronted with tasks like debugging, testing or modifying ASMs in case that we write ASM specifications. Such specifications might become very large and complex for real problems. Consequently, these three tasks possibly become very unprecise as the specifications are complex and hard to understand in detail. Slicing provides the possibility to observe only a specific, desired behavior rather than the global system behavior. Consequently, debugging, testing or modifying ASMs becomes a much easier task.

Already from the informal description in the introduction, it is easy to see that a minimal (static) slice cannot be computed automatically for unrestricted ASMs. However, a minimal slice is computable for ASMs from $\mathrm{GF}(\mathcal{D})$. The main goal of this part is to prove this proposition.

The outline of this part is as follows. First, we formally introduce the notion of slicing. The subsequent example of an ASM from $\mathrm{GF}(\mathcal{D})$ and some of its minimal slices is intended to increase the understanding of the notion of slice. It is rather easy to see that a minimal slice always exists but it is not unique. In the general

case, a minimal slice is not computable. The aim of the next two chapters is to prove that a minimal slice is computable for ASMs from GF($\mathcal{D}$). After having completed the proof, we analyze the complexity of the algorithm that can be derived from the proof. In the remainder of this part, we give some extensions of the basic result and present some variations for the notion of slicing.

At the end of this part on slicing, we briefly consider the equivalence of ASMs. It is much more straight to formulate an algorithm deciding equivalence of two ASMs from GF($\mathcal{D}$) than to compute a minimal slice though one might have the impression that these two problems are strongly related.

9.2 Definition of Slicing

In this chapter, we formally introduce the notion of slice for ASMs. At the beginning, we define the notion of slicing criterion characterizing those parts of a state in which we are interested. Afterwards, we formalize what it means to remove statements of an ASM in order to obtain a syntactically correct (and therefore executable) ASM again. Together with some notion of partial equivalence or more precisely some notion of equivalence relative to a slicing criterion, we formally introduce the notion of slice.

The state resp. the behavior of a program considered in [37] is defined by the values of its program variables. For ASMs, a state is defined by the content of its locations (see [20]). Locations can be described by atomic formulae.

Definition 9.1 (Slicing Criterion). A slicing criterion is a set of atomic formulae (possibly containing variables).

As in the classic case of program slicing, a slice of an ASM Π results from Π by removing parts from Π and resulting in a syntactically correct ASM again. The formal definition proceeds via the notion of subprogram.

Therefore, we introduce the notions of *subformula* of a first-order formula and *subrule* of an ASM. Note that they do not quite correspond to the usual definitions of these terms. The main point is that in order to obtain subformulae or subrules, we remove a part of the formula or program. The removed pieces might consist of more than one part, the parts are not necessarily connected and they can also be situated somewhere in the middle of the formula or the program. The formal definition of these notions is displayed in figure 9.1 and figure 9.2 where sub(ψ) abbreviates the set of subformulae of a FO-formula ψ and sub(Π) abbreviates the set of subrules of an ASM Π. A subrule without free variables is called a *subprogram*.

As a state of an ASM is defined by the contents of its locations, we want to be able to conclude from a state and a slicing criterion (and therefore from atomic

ψ	$\mathbf{sub}(\psi)$
atomic	$\{\psi, \text{true}, \text{false}\}$
$\neg\varphi$	$\{\psi\} \cup \text{sub}(\varphi) \cup \{\neg\vartheta : \vartheta \in \text{sub}(\varphi)\}$
$\varphi_1 \vee \varphi_2$	$\{\psi\} \cup \text{sub}(\varphi_1) \cup \text{sub}(\varphi_2) \cup \{\vartheta_1 \vee \vartheta_2 : \vartheta_1 \in \text{sub}(\varphi_1), \vartheta_2 \in \text{sub}(\varphi_2)\}$
$\varphi_1 \wedge \varphi_2$	$\{\psi\} \cup \text{sub}(\varphi_1) \cup \text{sub}(\varphi_2) \cup \{\vartheta_1 \wedge \vartheta_2 : \vartheta_1 \in \text{sub}(\varphi_1), \vartheta_2 \in \text{sub}(\varphi_2)\}$
$\exists \overline{x}\ \varphi$	$\{\psi\} \cup \text{sub}(\varphi) \cup \{\exists \overline{x}\ \vartheta : \vartheta \in \text{sub}(\varphi)\}$
$\forall \overline{x}\ \varphi$	$\{\psi\} \cup \text{sub}(\varphi) \cup \{\forall \overline{x}\ \vartheta : \vartheta \in \text{sub}(\varphi)\}$

Figure 9.1: Formal definition of $\text{sub}(\psi)$ for an FO-formula ψ

formulae) to the corresponding locations. The following definition yields this possibility.

Definition 9.2. Let $\mathcal{A}$ be a state and S be a slicing criterion.

For a location $l = (Q, a_1, \ldots, a_r)$ in $\mathcal{A}$, $\alpha(l)$ denotes the atom $Qa_1 \ldots a_r$.

$L(\mathcal{A}, S)$ denotes the set all locations l in $\mathcal{A}$ such that $\alpha(l)$ is an instance of an atomic formula of S in $\mathcal{A}$.

We define the notion of S-equivalence on a state $\mathcal{A}$ in order to be able to define S-equivalence in general where S is a slicing criterion.

Definition 9.3 (S-equivalence). Let Π and Π' be ASMs and S be a slicing criterion.

1. For a state $\mathcal{A}$ and a set L of locations of $\mathcal{A}$, let $\mathcal{A}|_L$ be the set of all pairs (l, a) where $l \in L$ and a is the content of l in $\mathcal{A}$.

2. Given a state $\mathcal{A}$ of Π resp. Π'. If for every run of Π $\mathcal{A} = \mathcal{A}_0 \vdash_\Pi \mathcal{A}_1 \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_i \vdash_\Pi \mathcal{A}_{i+1} \vdash_\Pi \ldots$ there is a run of Π' $\mathcal{A} = \mathcal{A}'_0 \vdash_{\Pi'} \mathcal{A}'_1 \vdash_{\Pi'} \cdots \vdash_{\Pi'} \mathcal{A}'_i \vdash_{\Pi'} \mathcal{A}'_{i+1} \vdash_{\Pi'} \ldots$ such that $\mathcal{A}_i|_{L(\mathcal{A}_i,S)} = \mathcal{A}'_i|_{L(\mathcal{A}'_i,S)}$ for all $i \in \mathbb{N}$ then Π' S-captures Π on $\mathcal{A}$ ($\Pi \leq^{\mathcal{A}}_S \Pi'$).

3. Given a state $\mathcal{A}$ of Π resp. Π'. Π and Π' are called S-equivalent on $\mathcal{A}$ ($\Pi \equiv^{\mathcal{A}}_S \Pi'$) if $\Pi \leq^{\mathcal{A}}_S \Pi'$ and $\Pi' \leq^{\mathcal{A}}_S \Pi$.

4. If $\Pi \equiv^{\mathcal{A}}_S \Pi'$ for all states $\mathcal{A}$ of Π resp. Π' then Π and Π' are called S-equivalent ($\Pi \equiv_S \Pi'$).

For deterministic ASMs, the notion of S-capturing and S-equivalence coincide.

Informally, one could formulate the condition for $\Pi \equiv_S \Pi'$ as follows. Π' behaves on all locations $(R, \overline{a})$ in exactly the same way as Π where the atom $R\overline{a}$ is an instance of an atomic formula $\alpha \in S$.

Now, we are ready to define the central notion of this part.

Π	$\mathbf{sub}(\Pi)$
Skip	$\{\mathbf{Skip}\}$
$s := t$	$\{\mathbf{Skip}, \Pi\}$
if φ **then** Π' **endif**	$\{\mathbf{Skip}, \Pi\} \cup \mathrm{sub}(\Pi') \cup$ $\{\mathbf{if}\ \vartheta\ \mathbf{then}\ \tilde{\Pi}\ \mathbf{endif} : \tilde{\Pi} \in \mathrm{sub}(\Pi'), \vartheta \in \mathrm{sub}(\varphi)\}$
do-in-parallel $\Pi_1\ \Pi_2$ **enddo**	$\{\mathbf{Skip}, \Pi\} \cup \mathrm{sub}(\Pi_1) \cup \mathrm{sub}(\Pi_2) \cup$ $\{\mathbf{do\text{-}in\text{-}parallel}\ \Pi_1'\ \Pi_2'\ \mathbf{enddo} :$ $\Pi_1' \in \mathrm{sub}(\Pi_1), \Pi_2' \in \mathrm{sub}(\Pi_2)\}$
import v R' **endimport**	$\{\mathbf{Skip}, \Pi\} \cup \mathrm{sub}(R') \cup$ $\{\mathbf{import}\ v\ \tilde{R}\ \mathbf{endimport} : \tilde{R} \in \mathrm{sub}(R')\}$
forall $\overline{x} : \varphi(\overline{x})$ **do** R' **endforall**	$\{\mathbf{Skip}, \Pi\} \cup \mathrm{sub}(R') \cup$ $\{\mathbf{forall}\ \overline{x} : \vartheta(\overline{x})\ \tilde{R}\ \mathbf{endforall} :$ $\tilde{R} \in \mathrm{sub}(R'), \vartheta \in \mathrm{sub}(\varphi)\}$
choose $\overline{x} : \varphi(\overline{x})$ **do** R' **endchoose**	$\{\mathbf{Skip}, \Pi\} \cup \mathrm{sub}(R') \cup$ $\{\mathbf{choose}\ \overline{x} : \vartheta(\overline{x})\ \tilde{R}\ \mathbf{endchoose} :$ $\tilde{R} \in \mathrm{sub}(R'), \vartheta \in \mathrm{sub}(\varphi)\}$

Figure 9.2: Formal definition of sub(Π) for an ASM Π

Definition 9.4 (Slice). Let Π be an ASM-program and S be a slicing criterion.
An S-slice of Π is an ASM-rule $\Pi_S \in \mathrm{sub}(\Pi)$ such that $\Pi \equiv_S \Pi_S$.

Remark. The set of locations, on which the both programs must behave in the same way, depends on the state. It is respectively generated by the state and the slicing criterion. If **import** is not used then this set depends only on the initial state of a run.

In the following, we refer to the size of a formula as the number of symbols in the string representing it where atomic formulae are counted as one. For a formula φ, $|\varphi|$ denotes its size.

The size of an ASM-rule is defined analogously but keywords are counted as one (then, endif, endforall, endchoose, enddo are not counted separately), update rules are counted as one and the size of the guards is defined by the above definition of the size of a formula. For an ASM Π, $|\Pi|$ denotes its size.

Formally, the size of an FO-formula or and ASM-rule is defined as follows.

Definition 9.5 (Size of an FO-formula). The size $|\psi|$ of an FO-formulae ψ is defined by induction on ψ:

1. $|\text{true}| = |\text{false}| = 0$
2. if ψ is an atom then $|\psi| = 1$
3. if $\psi = \neg\varphi$ then $|\psi| = 1 + |\varphi|$.
4. if $\psi = \varphi_1 \vee \varphi_2$ then $|\psi| = 1 + |\varphi_1| + |\varphi_2|$.
5. if $\psi = \varphi_1 \wedge \varphi_2$ then $|\psi| = 1 + |\varphi_1| + |\varphi_2|$.
6. if $\psi = \exists\overline{x}\varphi(\overline{x})$ then $|\psi| = 1 + |\varphi|$.
7. if $\psi = \forall\overline{x}\varphi(\overline{x})$ then $|\psi| = 1 + |\varphi|$.

Definition 9.6 (Size of an ASM-rule). The size $|R|$ of an ASM-rule R is defined by induction on the construction of the rule.

1. $|\textbf{Skip}| = 0$
2. If R is an update-rule of the form $s := t$ then $|R| = 1$.
3. if R is an if-clause of the form **if** α **then** R' **endif** then $|R| = 1 + |\alpha| + |R'|$.
4. if R is a block rule of the form **do-in-parallel** R_1 R_2 **enddo** then $|R| = 1 + |R_1| + |R_2|$.
5. If Π is an import rule of the form **import** v R' **endimport** then size$(\Pi) = 1 + |R|$.
6. if R is a forall rule of the form **forall** $\overline{x} : \alpha$ **do** R' **endforall** then $|R| = 1 + |\alpha| + |R'|$.

7. if R is a choice rule of the form **choose** $\overline{x}$: α **do** R' **endchoose** then $|R| = 1 + |\alpha| + |R'|$.

Note that in the following considerations, we could use any notion of size that is computable from an ASM or a formula.

Definition 9.7 (Minimal Slice). A minimal slice of an ASM-program Π and a slicing criterion S is an S-slice Π_S of Π such that there is no S-slice $\tilde{\Pi}$ of Π with $|\tilde{\Pi}| < |\Pi_S|$.

9.3 Example of a Minimal Slice

In this chapter, we exemplarily consider the ASM $\Pi \in \mathrm{GF}(\mathcal{D})$ (see figure 9.3). Essentially, an input of Π is a transition system with a set of distinguished nodes. The intended interpretations of the relations used in Π are the following.

For two states x and y of the transition system $\mathcal{T}$ and an action a, $Taxy$ holds if, and only if, a transition from x to y using a exists. P contains exactly the safe states of $\mathcal{T}$. S contains exactly the distinguished (starting) states of $\mathcal{T}$.

Initially, R is intended to be empty, liveness is equal to false and safe is equal to true. When the ASM halts (i.e., no more changes in the interpretations happen), R contains those states of $\mathcal{T}$ that are reachable from the set S via a finite sequence of actions. Furthermore, safe is true if, and only if, all reachable states are safe and liveness is true if, and only if, every reachable node has an outgoing edge.

It is not necessary to take into account the complete program in order to observe the evaluation of only a part of the relations. This observation is reflected by its slices.

Assume that Π is part of an ASM $\tilde{\Pi}$ (but the relations in Π are not updated outside Π). A wrong evaluation of R might be the reason for an error observed somewhere else in $\tilde{\Pi}$. Therefore, we are possibly interested only in that part of $\tilde{\Pi}$ that influences the evaluation of R. Consequently, it suffices to consider an $\{Rx\}$-slice of Π. There exists exactly one minimal $\{Rx\}$-slice (see figure 9.3). Though R influences the evaluation of safe and liveness, the other direction is not true. The evaluation of R does not depend on these relations. R depends only on the static unary relation S. Therefore, we can cut off all updates of other relations in order to obtain an $\{Rx\}$-slice of Π. Furthermore, it is not necessary to embed the update rule $Ry :=$ true in a conditional rule with guard $\neg Ry$. The rule **if** $\neg Ry$ **then** $Ry :=$ true **endif** is equivalent to the rule $Ry :=$ true. The reason for this equivalence is rather clear. We have to consider two cases. If Ry holds in a state then the update to true does not have any effect. Consequently, it does not make any difference whether we the update rule or the conditional rule is executed. If Ry does not hold then for both rules the update $Ry :=$ true is executed. However, it might be intuitive to embed the update rule in a conditional

```
do-in-parallel
  forall x : Sx do
    Rx := true
  endforall
  forall x : Rx do
    forall ay : Taxy do
      if ¬Ry then Ry := true endif
    endforall
  endforall
  forall x : Rx do
    do-in-parallel
      if ¬∃ay Taxy then liveness := false endif
      if ¬Px then safe := false endif
    enddo
  endforall
enddo
```

```
do-in-parallel
  forall x : Sx do
    Rx := true
  endforall
  forall x : Rx do
    forall ay : Taxy do
      Ry := true
    endforall
  endforall

enddo
```

Figure 9.3: ASM Π (left) and its minimal $\{Rx\}$-slice Π_R (right)

rule as the content of a location (R, a) is only changed if it is false in the current state.

If we are only interested in the evaluation of safe, we can restrict our considerations to a {safe}-slice of Π and there is exactly one minimal {safe}-slice. This slice is defined by the program

```
do-in-parallel
  forall x : Sx do
    Rx := true
  endforall
  forall x : Rx do
    forall ay : Taxy do
      Ry := true
    endforall
  endforall
  forall x : Rx do
    if ¬Px then safe := false endif
  endforall
enddo
```

From this slice, we see that a slice may involve updates of other locations whose left-hand side does not fit to an element of the slicing criterion. In this special case, the {safe}-slice contains updates of the relation R. It is clear that this can

not be avoided as the computation of safe strongly uses the relation R in a non-trivial way. Though safe and liveness are computed in parallel, the evaluation of safe does not involve liveness. Consequently, we can cut off all updates of liveness in order to obtain a {safe}-slice of Π.
Note that the minimal {safe}-slice is equal to the minimal {safe, Rx}-slice.

9.4 Existence, Computability and Non-Uniqueness

The existence of a minimal S-slice (of an ASM-program Π for a slicing criterion S) is obvious as sub(Π) is finite for any ASM Π and Π itself is an S-slice for any slicing criterion S. Note that the finiteness of sub(Π) does not imply that a minimal slice is computable. In general, it is not decidable whether an element of sub(Π) is a slice of Π. This can be proven rather easily.

Let φ be a first-order formula not containing the nullary relation symbol Mode. **Skip** is a minimal {Mode}-slice of Π = **if** φ **then** Mode := true **endif** if, and only if, φ is not satisfiable. Furthermore, **Skip** is the only minimal {Mode}-slice of Π in this case. Consequently, the undecidability of the finite satisfiability problem for first-order logic FO induces the non-computability of a minimal slice for deterministic ASMs (not using **import**).

Furthermore, a minimal slice is not unique. Consider for example the ASM defined by **do-in-parallel** R_1 R_2 **enddo** where the programs R_1 and R_2 are defined as follows.

R_1:

forall $xy : Rxy$ **do**
  do-in-parallel
    if$((P_1x \wedge P_2y) \vee P_3xy)$ **then** $Qxy :=$ true **endif**
    if$(P_2x \wedge P_1y)$ **then** $Qxy :=$ true **endif**
  enddo
endforall

R_2:

forall $xy : Rxy$ **do**
  do-in-parallel
    if $(P_1x \wedge P_2y)$ **then** $Qxy :=$ true **endif**
    if $((P_2x \wedge P_1y) \vee P_3xy)$ **then** $Qxy :=$ true **endif**
  enddo
endforall

where c and d are constant symbols.

R_1 and R_2 are equivalent and have the same size. Consequently, Π has two minimal $\{Qxy\}$-slices, namely R_1 and R_2 (which are different).

10 Quasistates and Partial Equivalence

The main goal of this part is an algorithm computing for a slicing criterion S and an ASM $\Pi \in \mathrm{GF}(\mathcal{D})$ a minimal S-slice of Π. This result uses the notion of quasistate. Essentially, every quasistate represents a class of states that evaluate certain formulae to the same values. The same behavior on all runs resp. S-equivalence is reduced to the same behavior on the runs of the standard states of the quasistates. In the proof of the computability result, the formulae occurring in the ASM are evaluated on quasistates. The idea of the notion of quasistate is similar to the one of quasimodel introduced in [3].

10.1 Quasistates

In this section, we define the notion of quasistate. The definitions in this section are mostly taken over or adapted from [3]. The notion of quasistate corresponds to the notion of quasimodel.

The formal definition of the notion of quasistate makes strongly use of the notion of type.

Definition 10.1 (Type). Let F be a finite set of FO-formulae. An F-type is a subset Δ of F such that

1. $\neg\psi \in \Delta$ if, and only if, not $\psi \in \Delta$ whenever $\neg\psi \in F$, and

2. $\psi \wedge \xi \in \Delta$ if, and only if, $\psi \in \Delta$ and $\xi \in \Delta$ whenever $\psi \wedge \xi \in F$, and

3. $[\overline{u}/\overline{y}]\psi$ implies $\exists \overline{y}\psi \in \Delta$ whenever $\exists \overline{y}\psi \in F$

where $[\overline{u}/\overline{y}]\psi$ is the formula obtained from ψ by replacing each free variable in $\overline{y}$ with the corresponding in $\overline{u}$, simultaneously.

A shorter but equivalent definition is the following. For a finite set F of FO-formulae, an F-type is a maximal consistent subset of F.

Definition 10.2. Let Δ be an F-type and V be the set of variables occurring in Δ. Let $\mathcal{A}$ be state.

$\mathcal{A}$ realizes Δ if there is a variable assignment $\sigma : V \to A$ (where A is the domain of $\mathcal{A}$) such that $\mathcal{A}, \sigma \models \varphi$ for all $\varphi \in \Delta$.

Definition 10.3 ($\overline{y}$-close). Let $\overline{y}$ be a sequence of variables, and let Δ, Δ' be types.

Δ and Δ' are $\overline{y}$-close ($\Delta =_{\overline{y}} \Delta'$) if Δ and Δ' have the same formulas with with free variables disjoint from $\overline{y}$.

Definition 10.4 (quasistate). An F-quasistate is a set of F-types T such that for each $\Delta \in T$ and each formula $\exists \overline{y} \varphi \in \Delta$ there is a type $\Delta' \in T$ with $\varphi \in \Delta'$ and $\Delta =_{\overline{y}} \Delta'$.

Φ holds in a quasistate if $\Phi \in \Delta$ for some Δ in this model.

In particular, the condition concerning the $\overline{y}$-closeness implies that for every sentence $\varphi \in F$ ($\text{free}(\varphi) = \emptyset$) $\varphi \in \Delta$ for all $\Delta \in Q$ or $\varphi \notin \Delta$ for all $\Delta \in Q$

For an F-quasistate Q, we define a standard state $\mathcal{A}_Q$.

π is a path $\pi = \langle \Delta_1, \Phi_1, \ldots, \Delta_n, \Phi_n \rangle$ if Δ_1 and Δ_{n+1} are types in Q, each formula Φ_i is of the form $\exists \overline{y} \varphi \in \Delta_i$, $\varphi \in \Delta_{i+1}$ and $\Delta_{i+1} \equiv_{\overline{y}} \Delta_i$.We say that the variables in $\overline{y}$ changed their values from Δ_i to Δ_{i+1} whereas the others did not. Furthermore, a variable z is called *new* in a path π if either $|\pi| = 1$ or the value of z was changed in the last round in π.

Now, we define $\mathcal{A}_Q$. The domain of $\mathcal{A}_Q$ consists of all pairs (π, z) where π is a path and z is new in π. In the rest of this paragraph, we give the interpretation of the predicates. For a relation R, $I(R)$ holds of the sequence of elements $\langle (\pi_j, x_j) \rangle_{j \in J}$ if, and only if, the paths π_j fit into one linear sequence under inclusion, with maximal path π^* such that Δ^* (the last type of π^*) contains $R\langle x_j \rangle_{j \in J}$ and x_j does not change its value on the further path to the end of π^* for any (π_j, x_j).

In the preceding part of this section, we have explained how one can construct for a quasistate Q a state $\mathcal{A}_Q$ realizing Q. Now, we consider the other direction. I.e., for a state $\mathcal{A}$, we give the maximal F-quasistate $Q^F_{\mathcal{A}}$ which is realized by $\mathcal{A}$. For an F-quasistate Q we say that the state $\mathcal{A}$ realizes Q if $\mathcal{A}$ realizes every $\Delta \in Q$.

Definition 10.5. Let F be a finite set of FO-formulae and $\mathcal{A}$ be a state over the same vocabulary. The F-quasistate of $\mathcal{A}$ is defined as:

$$Q^F_{\mathcal{A}} := \{\Delta : \Delta \text{ is an } F-\text{type which is realised in } \mathcal{A}\}$$

$Q^F_{\mathcal{A}}$ satisfies the conditions for an F-quasistate.

Definition 10.6. Let Q be an F-quasistate and V be the set of variables occurring in Q. A state $\mathcal{A}$ realizes Q if $\mathcal{A}$ realizes every $\Delta \in Q$.

10.2 S-equivalence on Quasistates

In this section, we prepare the proof of the computability result by giving some lemmas implying its proposition.

In order to shorten the formulations of the lemmas and the proofs, we introduce some terms.

First, we define for an ASM Π the set T_Π for every ASM. It essentially contains all formulae whose valuation possibly influences the run of an ASM Π.

Definition 10.7.
1. For an ASM Π, T_Π is the set of all FO-formulae over the vocabulary of Π such that their size is $\leq |\Pi|$ and only variables from Π are used.
2. For ASMs Π_1 and Π_2 let T_{Π_1,Π_2} denotes the set $T_{\Pi_1} \cup T_{\Pi_2}$.

The T_{Π_1,Π_2}-quasistate of a state $\mathcal{A}$ reflects the valuation in $\mathcal{A}$ of those formulae which may influence a run of Π_1 or Π_2 on $\mathcal{A}$. Therefore, we can restrict to the valuations of these formulae in order to reason about runs of Π. All other formulae do not have any influence. The proofs in the following will strongly make use of this quasistate.
$Q_{\mathcal{A}}^{\Pi_1,\Pi_2}$ abbreviates $Q_{\mathcal{A}}^{T_{\Pi_1,\Pi_2}}$ and $Q_{\mathcal{A}}^{\Pi}$ abbreviates $Q_{\mathcal{A}}^{T_\Pi}$.

In the previous section, we have defined the F-quasistate of a state $\mathcal{A}$. Analogously, one can define the F-type of a tuple in a state $\mathcal{A}$.

Definition 10.8. Let $\Pi \in \mathcal{D}$ be an ASM and $\mathcal{A}$ be a state of Π. Assume that the set of variables V of T_Π is ordered (i.e., $V = \{x_1, \ldots, x_n\}$).

Furthermore, let $\overline{a} = a_1 \ldots a_n$ be a tuple of elements from the domain A of $\mathcal{A}$. Then

1. $\overline{a}$ induces the variable assignment $\sigma_{\overline{a}} : V \to A$ such that $\sigma_{\overline{a}}(x_i) = a_i$ for $i \in \{1, \ldots, n\}$ (the variable assignment induced by $\overline{a}$).
2. the T_Π-type $\Delta_{\mathcal{A},\overline{a}}^{\Pi}$ induced by $\overline{a}$ in $\mathcal{A}$ is the T_Π-type $\Delta \in Q_{\mathcal{A}}^{\Pi}$ such that $\mathcal{A}, \sigma_{\overline{a}} \models \varphi$ for all $\varphi \in \Delta$ and $\mathcal{A}, \sigma_{\overline{a}} \models \neg\varphi$ for all $\varphi \in T_\Pi - \Delta$.

Remark. The type $\Delta_{\mathcal{A},\overline{a}}^{\Pi}$ is unique (w.l.o.g., we assume that a T_Π-state does not contain different alphabetical variants of a T_Π-type).

This can be easily proven by contradiction. Assume that there are two different T_Π-types Δ_1 and Δ_2 satisfying this condition.

As T_Π is closed under negation (for every formula $\varphi \in T_\Pi$ we also have $\neg\varphi \in T_\Pi$ where φ and $\neg\neg\varphi$ are identified), there is at least one formula $\varphi \in \Delta_1$ such that $\neg\varphi \in \Delta_2$ (by point 1. in definition 10.1).

By the assumption, we receive that $\mathcal{A}, \sigma_{\overline{a}} \models \varphi$ and $\mathcal{A}, \sigma_{\overline{a}} \models \neg\varphi$. This is a contradiction.

From [3], we know that the following proposition holds:

Lemma 10.9. A GF-formula Φ is satisfied by some state if, and only if, Φ holds in some quasistate.

It is obvious that if Φ is satisfied by some state then Φ is also satisfied by some quasistate. The main point in the proof of the converse is the following lemma.

Lemma 10.10. Let Q be an F-quasistate. For all paths π in $\mathcal{A}_Q$ and all formulae in F

$$\mathcal{A}_Q, s_\pi \models \psi \text{ if, and only if, } \psi \in \text{last}(\pi)$$

Lemma 10.11. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be states of an ASM $\Pi \in \mathcal{D}$, $\overline{a}$ be a tuple of elements from the domain of $\mathcal{A}_1$, $\overline{b}$ be a tuple of elements from the domain of $\mathcal{A}_2$ such that $\Delta^{\Pi}_{\mathcal{A}_1,\overline{a}} = \Delta^{\Pi}_{\mathcal{A}_2,\overline{b}}$. Furthermore, let $\mathcal{A}_1 \vdash_\Pi \mathcal{B}_1$ and $\mathcal{A}_2 \vdash_\Pi \mathcal{B}_2$.
Then $\Delta^{\Pi}_{\mathcal{B}_1,\overline{a}} = \Delta^{\Pi}_{\mathcal{B}_2,\overline{b}}$.

Proof. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be states of an ASM $\Pi \in \mathcal{D}$, $\overline{a}$ be a tuple of elements from the domain of $\mathcal{A}_1$, $\overline{b}$ be a tuple of elements from the domain of $\mathcal{A}_2$ such that $\Delta^{\Pi}_{\mathcal{A}_1,\overline{a}} = \Delta^{\Pi}_{\mathcal{A}_2,\overline{b}}$. Let $\mathcal{A}_1 \vdash_\Pi \mathcal{B}_1$ and $\mathcal{A}_2 \vdash_\Pi \mathcal{B}_2$.

Now, consider the T_Π-type $\Delta^{\Pi}_{\mathcal{B}_1,\overline{a}}$ of $\overline{a}$ in $\mathcal{B}_1$ resp. $\Delta^{\Pi}_{\mathcal{B}_2,\overline{b}}$ of $\overline{b}$ in $\mathcal{B}_2$ and the atoms α appearing in Δ. Every atom appearing in a formula in Δ is also contained in Δ, and therefore, the changes in the valuations $\sigma_1(\alpha)$, atom $\alpha \in \Delta$, determine the changes of all formulae in Δ and of those in $T_\Pi - \Delta$.

The valuation of $\sigma_{\overline{a}}(\alpha)$ is only changed if all formulae on the absolute guard (compare definition 13.2 to an update-rule $s := t$, where s fits to α, are satisfied in $\mathcal{A}_1$. As all formulae on this absolute guard and the boolean-valued terms occurring in Π_1 or Π_2 are contained in T_Π, they are satisfied in $\mathcal{A}_1, \sigma_{\overline{a}}$ if, and only if, they are satisfied in $\mathcal{A}_2, \sigma_{\overline{b}}$.

Therefore:
For every location $(R, \overline{u})$ in $\mathcal{A}_1$ resp. $(R, \overline{v})$ in $\mathcal{A}_2$ such that there is an atom $\alpha(\overline{x}) \in T_\Pi$ such that $\sigma_{\overline{a}}(\alpha(\overline{x})) = R\overline{u}$, $\sigma_{\overline{b}}(\alpha(\overline{x})) = R\overline{v}$, we get $\mathcal{B}_1, \sigma_{\overline{a}} \models R\overline{u}$ if, and only if, $\mathcal{B}_2, \sigma_{\overline{b}} \models R\overline{v}$.

Consequently, the T_Π-type induced by $\overline{a}$ in $\mathcal{B}_1$ and the T_Π-type induced by $\overline{b}$ in $\mathcal{B}_2$ are the same (up to alphabetical variants). In symbols, $\Delta^{\Pi}_{\mathcal{B}_1,\overline{a}} = \Delta^{\Pi}_{\mathcal{B}_2,\overline{b}}$. □

Corollary 10.12. Let $\mathcal{A}$ and $\mathcal{B}$ be states of an ASM $\Pi \in \mathcal{D}$, $\overline{a}$ be a tuple of elements from the domain of $\mathcal{A}$, $\overline{b}$ be a tuple of elements from the domain of $\mathcal{B}$ such that $\Delta^{\Pi}_{\mathcal{A},\overline{a}} = \Delta^{\Pi}_{\mathcal{B},\overline{b}}$. Furthermore, let $\mathcal{A} = \mathcal{A}_0 \vdash_\Pi \mathcal{A}_1 \vdash_\Pi \mathcal{A}_2 \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_i \vdash_\Pi \ldots$ resp. $\mathcal{B} = \mathcal{B}_0 \vdash_\Pi \mathcal{B}_1 \vdash_\Pi \mathcal{B}_2 \vdash_\Pi \cdots \vdash_\Pi \mathcal{B}_i \vdash_\Pi \ldots$ the run of Π on $\mathcal{A}$ resp. $\mathcal{B}$.
Then, for all $i \in \mathbb{N}$ $\Delta^{\Pi}_{\mathcal{A}_i,\overline{a}} = \Delta^{\Pi}_{\mathcal{B}_i,\overline{b}}$.

Proof. This corollary is a direct consequence of lemma 10.11.

Let $\mathcal{A}$ and $\mathcal{B}$ be states of an ASM $\Pi \in \mathcal{D}$, $\overline{a}$ be a tuple of elements from the domain of $\mathcal{A}$, $\overline{b}$ be a tuple of elements from the domain of $\mathcal{B}$ such that

$\Delta^{\Pi}_{\mathcal{A},\overline{a}} = \Delta^{\Pi}_{\mathcal{B},\overline{b}}$. Furthermore, let $\mathcal{A} = \mathcal{A}_0 \vdash_\Pi \mathcal{A}_1 \vdash_\Pi \mathcal{A}_2 \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_i \vdash_\Pi \ldots$ resp. $\mathcal{B} = \mathcal{B}_0 \vdash_\Pi \mathcal{B}_1 \vdash_\Pi \mathcal{B}_2 \vdash_\Pi \cdots \vdash_\Pi \mathcal{B}_i \vdash_\Pi \ldots$ the run of Π on $\mathcal{A}$ resp. $\mathcal{B}$.
The proof can be done by induction on i.

$i = 0$: In this case, the proposition is obvious as $\Delta^{\Pi}_{\mathcal{A},\overline{a}} = \Delta^{\Pi}_{\mathcal{B},\overline{b}}$ by assumption.

$i \to i+1$: Assume that the proposition has already been proven for $i \in \mathbb{N}$. Therefore, $\Delta^{\Pi}_{\mathcal{A}_i,\overline{a}} = \Delta^{\Pi}_{\mathcal{B}_i,\overline{b}}$. Using lemma 10.11, we get that $\Delta^{\Pi}_{\mathcal{A}_{i+1},\overline{a}} = \Delta^{\Pi}_{\mathcal{B}_{i+1},\overline{b}}$.

By induction, we have proven the corollary. □

The following lemma provides the possibility to conclude from standard states to arbitrary states.

Lemma 10.13. Let $\Pi_1, \Pi_2 \in \mathcal{D}$ and Q be a T_{Π_1,Π_2}-quasistate. Furthermore, let $\mathcal{A}_Q$ be the standard state of Q and S be a slicing criterion. If $\Pi_1 \equiv_S^{\mathcal{A}_Q} \Pi_2$ then $\Pi_1 \equiv_S^{\mathcal{A}} \Pi_2$ for all states $\mathcal{A}$ with $Q_{\mathcal{A}}^{\Pi_1,\Pi_2} = Q$.

Proof. Let $\Pi_1, \Pi_2 \in \mathcal{D}$ and Q be a T_{Π_1,Π_2}-quasistate. Furthermore, let $\mathcal{A}_Q$ be the standard state of Q and S be a slicing criterion. W.l.o.g., we can assume that $S \subseteq T_{\Pi_1,\Pi_2}$ as locations $(R, \overline{a})$ where $R\overline{a}$ is not an instance of an element of T_{Π_1,Π_2} are static for Π_1 and for Π_2. Consequently, Π_1 and Π_2 behave on these locations in the same way.

Let $\mathcal{A}$ be any state whose T_{Π_1,Π_2}-quasistate is Q and let $\mathcal{A} = \mathcal{A}_0 \vdash_{\Pi_1} \mathcal{A}_1 \vdash_{\Pi_1} \mathcal{A}_2 \vdash_{\Pi_1} \cdots \vdash_{\Pi_1} \mathcal{A}_i \vdash_{\Pi_1} \ldots$ resp. $\mathcal{A} = \mathcal{B}_0 \vdash_{\Pi_2} \mathcal{B}_1 \vdash_{\Pi_2} \mathcal{B}_2 \vdash_{\Pi_2} \cdots \vdash_{\Pi_2} \mathcal{B}_i \vdash_{\Pi_2} \ldots$ be the run of Π_1 resp. Π_2 on $\mathcal{A}$.

For any n-tuple $\overline{a}$ (where n is the number of variables occurring in T_{Π_1,Π_2}) of elements from the domain of $\mathcal{A}$, there is a type $\Delta^{\Pi_1,\Pi_2}_{\mathcal{A},\overline{a}} \in Q$ that contains exactly those formulae $\varphi \in T_{\Pi_1,\Pi_2}$ with $\mathcal{A}, \sigma_{\overline{a}} \models \varphi$ ($\sigma_{\overline{a}}$ is the variable assignment induced by $\overline{a}$, see above). According to the definition of $\mathcal{A}_Q$, there is an n-tuple $\overline{b}$ of elements from the domain of $\mathcal{A}_Q$ (the standard state for Q) such that $\mathcal{A}_Q, \sigma_{\overline{b}} \models \varphi$ holds for exactly those formulae $\varphi \in T_{\Pi_1,\Pi_2}$ that are contained in $\Delta^{\Pi_1,\Pi_2}_{\mathcal{A},\overline{a}}$. Consequently, $\overline{b}$ realizes $\Delta^{\Pi_1,\Pi_2}_{\mathcal{A},\overline{a}}$ in $\mathcal{A}_Q$.

Using corollary 10.12, we obtain that $\Delta^{\Pi_1}_{\mathcal{A}_i,\overline{a}} = \Delta^{\Pi_1}_{\mathcal{A}_i^Q,\overline{b}}$ and $\Delta^{\Pi_2}_{\mathcal{B}_i,\overline{a}} = \Delta^{\Pi_2}_{\mathcal{A}_i^Q,\overline{b}}$ for all $i \in \mathbb{N}$. Assuming that $\Pi_1 \equiv_S^{\mathcal{A}_Q} \Pi_2$, this implies that $\Delta^{\Pi_1,\Pi_2}_{\mathcal{A}_i,\overline{a}} = \Delta^{\Pi_1,\Pi_2}_{\mathcal{B}_i,\overline{a}}$ for all $i \in \mathbb{N}$.

As $\overline{a}$ has been chosen arbitrarily, the above argumentation holds for any n-tuple of elements from the domain of $\mathcal{A}$. Consequently, we obtain that for all $i \in \mathbb{N}$ $\mathcal{A}_i|_{L(\mathcal{A}_i,S)} = \mathcal{B}_i|_{L(\mathcal{B}_i,S)}$. By definition, this is equivalent to $\Pi_1 \equiv_S^{\mathcal{A}} \Pi_2$. □

Lemma 10.14. Let $\Pi_1, \Pi_2 \in \mathrm{GF}(\mathcal{D})$. If for all $T_{\Pi_1,\Pi_2} \cap \mathrm{GF}$-quasistates Q $\Pi_1 \equiv_S^{\mathcal{A}_Q} \Pi_2$ then $\Pi_1 \equiv_S \Pi_2$.

Proof. For two ASMs $\Pi_1, \Pi_2 \in \mathrm{GF}(\mathcal{D})$, the set T_{Π_1,Π_2} is a subset of GF (the guarded fragment of first-order logic). Using lemma 10.9 and the property that GF is closed under negation, a formula $\Phi \in T_{\Pi_1,\Pi_2}$ is satisfied in all T_{Π_1,Π_2}-quasistates if, and only if, it is satisfied in all states over the same vocabulary. As a formula is satisfied in the T_{Π_1,Π_2}-quasistate Q if, and only if, it is satisfied in $\mathcal{A}_Q$, a formula Φ is satisfied in all states if, and only if, it is satisfied in the standard state $\mathcal{A}_Q$ for all T_{Π_1,Π_2}-quasistates Q.

Therefore, this lemma is a direct consequence of lemma 10.13. □

Note that this is the point where we need to introduce the restriction to the guarded fragment. The reason is that the proposition from [3] does not hold for arbitrary FO-formula but for formulae from GF. For ASMs Π_1, Π_2 from $\mathrm{GF}(\mathcal{D})$, we can restrict our considerations to the subset $T_{\Pi_1,\Pi_2} \cap \mathrm{GF}$ of T_{Π_1,Π_2}

11 Computing a Minimal Slice

11.1 Computability of a Minimal Slice

Theorem 11.1. There exists an algorithm computing for a slicing criterion S and an ASM $\Pi \in \mathrm{GF}(\mathcal{D})$ a minimal S-slice of Π.

Proof. From lemma 10.14 we know the following. In order to check for two ASMs Π_1 and Π_2 from $\mathrm{GF}(\mathcal{D})$, whether $\Pi_1 \equiv_S \Pi_2$, it is sufficient to check whether $\Pi_1 \equiv_S^{\mathcal{A}_Q} \Pi_2$ for all T_{Π_1,Π_2}-quasistates Q.

For a T_{Π_1,Π_2}-quasistate Q, one can check whether $\Pi_1 \equiv_S^{\mathcal{A}_Q} \Pi_2$ as follows.

As the domain A_Q of $\mathcal{A}_Q$ is finite, there is only a finite number a_q of reachable states over A_Q. a_q is computable from the vocabulary of $\mathcal{A}_Q$ and the size of its domain. More precisely, a_Q can be assumed to be $\prod_{R/n_R \in q\Upsilon \text{ is a relation symbol}} 2^{|A_Q|^{n_R}}$ where A_Q is the domain of $\mathcal{A}_Q$ and Υ is the vocabulary of Π_1 resp. Π_2. Note that constant symbols do not need to be taken into account as their interpretation can not be changed by an ASM from $\mathcal{D}$.

To check whether $\Pi_1 \equiv_S^{\mathcal{A}_Q} \Pi_2$, construct the run of Π_1 up to the length $a_q + 1$ and the respective initial segment of the run of Π_2. This leads to $\mathcal{A}_Q = \mathcal{A}_0 \vdash_{\Pi_1} \mathcal{A}_1 \vdash_{\Pi_1} \mathcal{A}_2 \vdash_{\Pi_1} \cdots \vdash_{\Pi_1} \mathcal{A}_i \vdash_{\Pi_1} \cdots \vdash_{\Pi_1} \mathcal{A}_{a_Q+1}$ and $\mathcal{A} = \mathcal{B}_0 \vdash_{\Pi_2} \mathcal{B}_1 \vdash_{\Pi_2} \mathcal{B}_2 \vdash_{\Pi_2} \cdots \vdash_{\Pi_2} \mathcal{B}_i \vdash_{\Pi_2} \cdots \vdash_{\Pi_2} \mathcal{B}_{a_Q+1}$. Now, check whether $\mathcal{A}_i|_{L(\mathcal{A}_i,S)} = \mathcal{B}_i|_{L(\mathcal{B}_i,S)}$ for all $i \in \{1, \ldots, a_Q + 1\}$.

For two ASMs Π_1 and Π_2, the set of $(T_{\Pi_1,\Pi_2} \cap \mathrm{GF})$-quasistates is finite. Therefore, it is decidable whether $\Pi_1 \equiv_S \Pi_2$ for $\Pi_1, \Pi_2 \in \mathrm{GF}(\mathcal{D})$. This is a consequence of lemma 10.14.

As $\mathrm{sub}(\Pi)$ is finite for every ASM Π and computable from Π, we are now able to formulate the following algorithm computing for a slicing criterion S and an ASM Π' a minimal S-slice of Π.

For every program $\Pi' \in \mathrm{sub}(\Pi)$, check whether $\Pi \equiv_S \Pi'$. Return a minimal ASM from $\mathrm{sub}(\Pi)$ satisfying this condition. □

11.2 Worst-Case Complexity

It is known that the satisfiability and validity problem for the guarded fragment of first-order logic are complete for 2EXPTIME. Furthermore, for an ASM **if**φ **then** Mode := true **endif**, the ASM Mode := true is a {Mode}-slice if, and only if, $\neg\varphi$ is not satisfiable. This implies that deciding whether an ASM $\Pi_1 \in \mathrm{GF}(\mathcal{D})$ is an S-slice of an ASM $\Pi_2 \in \mathrm{GF}(\mathcal{D})$ is hard for 2EXPTIME. Consequently, we can not expect a complexity below 2EXPTIME for computing a minimal slice of an ASM from $\mathrm{GF}(\mathcal{D})$.

In order to determine the complexity of the above slicing algorithm, we first consider the size of $\mathrm{sub}(\psi)$ for an FO-formula ψ and the size of $\mathrm{sub}(\Pi)$ for an ASM Π.

Corollary 11.2. Let ψ be an FO-formula. Then $|\mathrm{sub}(\psi)| \leq 2^{2|\psi|}$.

Proof. We prove the corollary by induction on ψ.

- If ψ is atomic then $|\mathrm{sub}(\psi)| = 1 \leq 2^2 = 2^{2|\psi|}$
- If $\psi = \neg\varphi$ then
  $|\mathrm{sub}(\psi)| = 1 + 2 \cdot |\mathrm{sub}(\varphi)| \leq 1 + 2 \cdot 2^{2|\varphi|} \leq 2^2 \cdot 2^{2|\varphi|} = 2^{2(1+|\varphi|)} = 2^{2|\psi|}$
- If $\psi = \varphi_1 \vee \varphi_2$ then

$$\begin{aligned} |\mathrm{sub}(\psi)| &\leq 1 + |\mathrm{sub}(\varphi_1)| + |\mathrm{sub}(\varphi_2)| + |\mathrm{sub}(\varphi_1)| \cdot |\mathrm{sub}(\varphi_2)| \\ &\leq 1 + 2^{2|\varphi_1|} + 2^{2|\varphi_2|} + 2^{2(|\varphi_1|+|\varphi_2|)} \\ &\leq 1 + 2 \cdot 2^{2(|\varphi_1|+|\varphi_2|)} \\ &\leq 2^2 \cdot 2^{2(|\varphi_1|+|\varphi_2|)} \\ &\leq 2^{2(1+|\varphi_1|+|\varphi_2|)} \\ &= 2^{2|\psi|} \end{aligned}$$

- If $\psi = \varphi_1 \wedge \varphi_2$ then we obtain a computation similar to the one for $\psi = \varphi_1 \vee \varphi_2$.
- If $\psi = \exists\overline{x}\ \varphi$ then

$$\begin{aligned} |\mathrm{sub}(\psi)| &\leq 1 + 2 \cdot |\mathrm{sub}(\varphi)| \\ &\leq 1 + 2 \cdot 2^{2|\varphi|} \\ &\leq 2^2 \cdot 2^{2|\varphi|} \\ &\leq 2^{2(1+|\varphi|)} \\ &= 2^{2|\psi|} \end{aligned}$$

- If $\psi = \forall\overline{x}(\alpha(\overline{x},\overline{y}) \wedge \varphi)$ then we obtain a computation similar to the one for $\psi = \exists\overline{x}(\alpha(\overline{x},\overline{y}) \wedge \varphi)$.

By induction, the proposition of the corollary follows. □

Corollary 11.3. Let $\mathcal{C}$ be the class of all ASM and $\Pi \in \mathrm{GF}(\mathcal{C})$. Then $|\mathrm{sub}(\Pi)| \leq 2^{2|\Pi|}$

Proof. We prove the corollary by induction on Π.

- If Π is an update rule then $|\mathrm{sub}(\Pi)| = 1 \leq 2^2 = 2^{2|\Pi|}$.
- If $\Pi =$ **if** φ **then** Π' **endif** then

$$\begin{aligned} |\mathrm{sub}(\Pi)| &\leq 2 + |\mathrm{sub}(\Pi')| + |\mathrm{sub}(\varphi)| \cdot |\mathrm{sub}(\Pi')| \\ &\leq 2 + 2^{2|\Pi'|} + 2^{2|\varphi|} \cdot 2^{2|\Pi'|} \\ &= 2 + 2^{2|\Pi'|} + 2^{2(|\varphi|+|\Pi'|)} \\ &\leq 2^2 \cdot 2^{2(|\varphi|+|\Pi'|)} \\ &= 2^{2(1+|\varphi|+|\Pi'|)} \\ &= 2^{2|\Pi|} \end{aligned}$$

- If $\Pi =$ **do-in-parallel** Π_1 Π_2 **enddo** then

$$\begin{aligned} |\mathrm{sub}(\Pi)| &\leq 2 + |\mathrm{sub}(\Pi_1)| + |\mathrm{sub}(\Pi_2)| + |\mathrm{sub}(\Pi_1)| \cdot |\mathrm{sub}(\Pi_2)| \\ &\leq 2 + 2^{2|\Pi_1|} + 2^{2|\Pi_2|} + 2^{2|\Pi_1|} \cdot 2^{2|\Pi_2|} \\ &\leq 2 + 2^{2|\Pi_1|} + 2^{2|\Pi_2|} + 2^{2(|\Pi_1|+|\Pi_2|)} \\ &\leq 2^2 \cdot 2^{2(|\Pi_1|+|\Pi_2|)} \\ &= 2^{2(1+|\Pi_1|+|\Pi_2|)} \\ &= 2^{2|\Pi|} \end{aligned}$$

- If $\Pi =$ **import** v R **endimport** then

$$\begin{aligned} |\mathrm{sub}(\Pi)| &\leq 2 + 2 \cdot |\mathrm{sub}(R)| \\ &\leq 2 + 2 \cdot 2^{2|R|} \\ &\leq 2^2 \cdot 2^{2|R|} \\ &= 2^{2(|R|+1)} \\ &= 2^{2|\Pi|} \end{aligned}$$

- If $\Pi =$ **forall** $\overline{x} : \alpha(\overline{x})$ R **endforall** then

$$\begin{aligned} |\mathrm{sub}(\Pi)| &\leq 2 + |\mathrm{sub}(R)| + |\mathrm{sub}(\alpha)| \cdot |\mathrm{sub}(R)| \\ &\leq 2 + 2^{2|R|} + 2^{2|\alpha|} \cdot 2^{2|R|} \\ &\leq 2 + 2 \cdot 2^{2|\alpha|} \cdot 2^{2|R|} \\ &\leq 2 \cdot 2 \cdot 2^{2|\alpha|} \cdot 2^{2|R|} \\ &= 2^{2(1+|\alpha|+|R|)} \\ &= 2^{2|\Pi|} \end{aligned}$$

- If $\Pi =$ **choose** $\overline{x} : \alpha(\overline{x})\ R$ **endchoose** then we obtain a computation similar to the case $\Pi =$ **forall** $\overline{x} : \alpha(\overline{x})\ R$ **endforall**.

By induction, the proposition of the corollary follows. □

Via simple combinatorial considerations, we get the following corollary.

Corollary 11.4. Let Π be an ASM over vocabulary Υ containing only constant and relation symbols. Furthermore, let V be the set of all variables in Π, $\sharp_{rel}$ be the number of relation symbols and max be the maximal arity of relation symbols in Υ.

Then the number of atomic formulae over Υ and using only variables from V is $\leq \sharp_{rel} \cdot (|V| + |\Upsilon| - \sharp_{rel})^{\max}$.

As a consequence of the above corollary, we get that $|T_\Pi| \in O(2^{|\Pi|^2 \cdot \log |\Pi|})$ for any ASM Π.
As the set of all T_Π-types is an element of the potential set of T_Π, we can conclude that the number of T_Π-types is $\leq 2^{|T_\Pi|} \leq 2^{2^{|\Pi|^2 \cdot \log |\Pi|}}$.
As the set of T_Π-quasistates is an element of the potential set of the set of all T_Π-types, we get that the number of T_Π-quasistates is $\leq 2^{2^{2^{|\Pi|^2 \cdot \log |\Pi|}}}$.
For all paths π in a quasistate Q, $|\pi| \leq \max\{|\Delta| : \Delta \in Q\}$. Therefore, The size of the domain of $\mathcal{A}_Q$ (the standard state of Q) is $\leq |Q|^2$. Therefore, the maximal number of states over the domain of $\mathcal{A}_Q$ is $\leq \Pi_{R \in \Upsilon} 2^{|Q|^{2 \cdot \text{arity}(R)}}$. The number a_Q in the proof of theorem 11.1 can therefore be set to $\leq \Pi_{R \in \Upsilon} 2^{|Q|^{2 \cdot \text{arity}(R)}}$.

As a consequence of the above considerations is that the algorithm derived from the proof of theorem 11.1 needs at most time in $O(2^{2^{2p(|\Pi|)}})$ for some polynomial p to compute a minimal slice for an ASM $\Pi \in \text{GF}(\mathcal{D})$.

11.3 Practical Complexity

Note that the above complexities are those for the worst-case. For practical applications we would be more interested in the average-case complexity which is possibly much lower and even acceptable for practical purposes. But this one depends strongly on the applications and may differ for different ones such that it is not possible to give a uniform average-case complexity.

Moreover, the runtime of the algorithm in practice depends strongly on the concrete implementation of the algorithm. There is a number of optimizations which do not change the runtime in the worst-case but improve it in many cases. We give some examples in the following. First of all, one could try all subprogram increasingly ordered by size. Furthermore, the test of a subprogram can be broken off as soon as there is a difference in a run (relative to the slicing criterion) to the respective run of the original program. While constructing the run of the subprogram, we can directly check this. There are many programs

and subprograms where such a difference arises rather early. A third possibility of optimization arises from the observation that the size of T_Π strongly increases with the number of variables used in the program Π. But if e.g. two distinct variables are bounded by independent forall-rules then they could be substituted by the same variable without changing the semantics of the original ASM. This is e.g. the case if one has a program of the form **do-in-parallel** Π_1 Π_2 **enddo** where Π_1 and Π_2 are forall-rules binding distinct variables. Consequently, one could optimize an implementation of the algorithm by preprocessing the input program such that its semantics is not changed but the number of distinct variables is lowered.

11.4 Extensions of the Basic Result

11.4.1 Extension by Nondeterminism

Having considered slicing for ASMs from GF($\mathcal{D}$), we arrive naturally at the question whether it is also possible to compute a minimal slice for an ASM from GF($\mathcal{ND}$). The answer is positive.

Theorem 11.5. There exists an algorithm computing for a slicing criterion S and an ASM $\Pi \in \mathrm{GF}(\mathcal{ND})$ a minimal S-slice of Π.

The proof of the computability of a minimal slice for nondeterministic ASMs from GF($\mathcal{ND}$) proceeds in the same steps as the one for deterministic ASMs and the proofs in these steps are rather similar to those in the deterministic case. The only difference is that instead of runs we have to consider computation graphs where a state may have more than one successor. In this extended scenario, the same argumentation as for GF($\mathcal{D}$) can be used except that we do not reason about *the* successor of a state but about one successor and the existence of another one in the run of a subprogram with certain properties.

Lemma 10.11, corollary 10.12, lemma 10.13, lemma 10.14 have to be transfered to the nondeterministic case as follows.

Lemma 11.6. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be states of an ASM $\Pi \in \mathcal{ND}$, $\overline{a}$ be a tuple of elements from the domain of $\mathcal{A}_1$, $\overline{b}$ be a tuple of elements from the domain of $\mathcal{A}_2$ such that $\Delta^\Pi_{\mathcal{A}_1,\overline{a}} = \Delta^\Pi_{\mathcal{A}_2,\overline{b}}$. Furthermore, let $\mathcal{A}_1 \vdash_\Pi \mathcal{B}_1$.
Then there exists a state $\mathcal{B}_2$ such that $\mathcal{A}_2 \vdash_\Pi \mathcal{B}_2$ and $\Delta^\Pi_{\mathcal{B}_1,\overline{a}} = \Delta^\Pi_{\mathcal{B}_2,\overline{b}}$.

Corollary 11.7. Let $\mathcal{A}$ and $\mathcal{B}$ be states of an ASM $\Pi \in \mathcal{ND}$, $\overline{a}$ be a tuple of elements from the domain of $\mathcal{A}$, $\overline{b}$ be a tuple of elements from the domain of $\mathcal{B}$ such that $\Delta^\Pi_{\mathcal{A},\overline{a}} = \Delta^\Pi_{\mathcal{B},\overline{b}}$. Furthermore, let $\mathcal{A} = \mathcal{A}_0 \vdash_\Pi \mathcal{A}_1 \vdash_\Pi \mathcal{A}_2 \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_i \vdash_\Pi \ldots$ a run of Π on $\mathcal{A}$.
Then, there exists a run $\mathcal{B} = \mathcal{B}_0 \vdash_\Pi \mathcal{B}_1 \vdash_\Pi \mathcal{B}_2 \vdash_\Pi \cdots \vdash_\Pi \mathcal{B}_i \vdash_\Pi \ldots$ of Π on $\mathcal{B}$ such that for all $i \in \mathbb{N}$ $\Delta^\Pi_{\mathcal{A}_i,\overline{a}} = \Delta^\Pi_{\mathcal{B}_i,\overline{b}}$.

Lemma 11.8. Let $\Pi_1, \Pi_2 \in \mathcal{ND}$ and Q be a T_{Π_1,Π_2}-quasistate. Furthermore, let $\mathcal{A}_Q$ be the standard state of Q and S be a slicing criterion.
If $\Pi_1 \equiv_S^{\mathcal{A}_Q} \Pi_2$ then $\Pi_1 \equiv_S^{\mathcal{A}} \Pi_2$ for all states $\mathcal{A}$ with $Q_{\mathcal{A}}^{\Pi_1,\Pi_2} = Q$.

Lemma 11.9. Let $\Pi_1, \Pi_2 \in \mathrm{GF}(\mathcal{ND})$. If for all $T_{\Pi_1,\Pi_2} \cap \mathrm{GF}$-quasistates Q $\Pi_1 \equiv_S^{\mathcal{A}_Q} \Pi_2$ then $\Pi_1 \equiv_S \Pi_2$.

11.4.2 Extension of Deterministic ASMs by import

Having extended the slicing result by nondeterminism we consider the question whether it is also possible to compute a minimal slice for an ASM from $\mathrm{GF}(\mathcal{DI})$. The answer is again positive.

Theorem 11.10. There exists an algorithm computing for a slicing criterion S and an ASM $\Pi \in \mathrm{GF}(\mathcal{DI})$ a minimal S-slice of Π.

The proof of the computability of a minimal slice for deterministic ASMs from $\mathrm{GF}(\mathcal{DI})$ proceeds similar to the one for deterministic ASMs from $\mathrm{GF}(\mathcal{D})$.

Analogously to the case without **import**, we consider the runs on the standard states but here, we modify the obtained structure after every step such that we do not get the real runs but a kind of reduction of them. The reason for the need of such a reduction is that the active domain of the states might grow grow in each execution step and therefore, the argument that there are only finitely many states over the domain of the initial state does not work anymore. After each computation step, the set of imported elements is reduced to a representative set of limited size. We do not give the complete proof of theorem 11.10 but we describe the most important modifications in the proof of theorem 11.1.

W.l.o.g., we assume that if an ASM-rule R appears inside an import rule (i.e., in the form **import** v R **endimport**) then v appears only in the left-hand side of an update rule in R.

Definition 11.11. Let $\mathcal{A}$ be a state and a be an element from the domain of $\mathcal{A}$.

The atomic type of a in $\mathcal{A}$ (in symbols $\mathrm{at}(a, \mathcal{A})$) is the set of all atoms $R\overline{u}$ (where $\overline{u} = u_1 \dots u_n$ and for $i \in \{1, \dots, n\}$ u_i is an element the domain of $\mathcal{A}$ or the variable x) such that $\mathcal{A} \models R\overline{u}[x/a]$.

Definition 11.12. Let $\mathcal{A}$ be a state, a and b be elements from the domain of $\mathcal{A}$.

$a \sim_{\mathcal{A}} b$ if, and only if, $\mathrm{at}(a, \mathcal{A}) = \mathrm{at}(b, \mathcal{A})$.

Lemma 11.13. Let $\Pi \in \mathrm{GF}(\mathcal{DI})$ be an ASM, $\mathcal{A}$ be a state and A be the domain of $\mathcal{A}$. Furthermore, let $\mathcal{A} = \mathcal{A}_0 \vdash_\Pi \mathcal{A}_1 \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_i \vdash_\Pi \mathcal{A}_{i+1} \vdash_\Pi \dots$ and $a, b \in A$ such that $\mathcal{A} \models \mathrm{reserve}(a)$ and $\mathcal{A} \models \mathrm{reserve}(b)$. Furthermore, let i be the smallest number such that $\mathcal{A}_i \models \neg\mathrm{reserve}(a)$ and j be the smallest number $\mathcal{A}_j \models \neg\mathrm{reserve}(b)$. Let $k = \max(i, j)$.

If $\mathrm{at}(a, \mathcal{A}_k) = \mathrm{at}(b, \mathcal{A}_k)$ then $\mathrm{at}(a, \mathcal{A}_l) = \mathrm{at}(b, \mathcal{A}_l)$ for all $l \geq k$.

Proof. Let $\Pi \in \mathrm{GF}(\mathcal{DI})$ be an ASM, $\mathcal{A}$ be a state and A be the domain of $\mathcal{A}$. Furthermore, let $\mathcal{A} = \mathcal{A}_0 \vdash_\Pi \mathcal{A}_1 \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_i \vdash_\Pi \mathcal{A}_{i+1} \vdash_\Pi \ldots$ and $a, b \in A$ such that $\mathcal{A} \models \mathrm{reserve}(a)$ and $\mathcal{A} \models \mathrm{reserve}(b)$. Furthermore, let i be the smallest number such that $\mathcal{A}_i \models \neg\mathrm{reserve}(a)$ and j be the smallest number $\mathcal{A}_j \models \neg\mathrm{reserve}(b)$. Let $k = \max(i, j)$. Assume that $\mathrm{at}(a, \mathcal{A}_k) = \mathrm{at}(b, \mathcal{A}_k)$.

The atomic type of a resp. b can only be changed inside a forall-rule **forall** $\overline{y}$: $\alpha(\overline{y})$ R **endforall** where R contains an update rule $R\overline{z} := t$ ($\overline{z} = z_1 \ldots z_n$ such that z_i is a variable or a constant symbol from Υ for $i \in \{1, \ldots, n\}$).

Assume that the atomic type of a is changed. W.l.o.g., there is an atom $R\overline{u} \in \mathrm{at}(a, \mathcal{A}_m)$ such that $R\overline{u}[x/a]$ fits to $R\overline{z}$ in $\mathcal{A}$ and $\mathrm{conj}(\mathrm{absGuard}(s := t, R)) \wedge \alpha(\overline{y})[\overline{z}/\overline{u}][x/a]$ is satisfied and $t[\overline{z}/\overline{u}][x/a]$ is not satisfied. As $\mathrm{at}(a, \mathcal{A}_m) = \mathrm{at}(b, \mathcal{A}_m)$, $t[\overline{z}/\overline{u}][x/a]$ is not satisfied in $\mathcal{A}_m$ and the above holds for all update rules in R, we get $\mathrm{at}(a, \mathcal{A}_{m+1}) = \mathrm{at}(b, \mathcal{A}_{m+1})$

By induction, we obtain that $\mathrm{at}(a, \mathcal{A}_l) = \mathrm{at}(b, \mathcal{A}_l)$ for all $l \geq k$. □

In the remainder of this section we sketch what we have to change in the proof of theorem 11.1 in order to obtain a proof of theorem 11.10.

The first thing that we have to notice for extending theorem 11.1 by **import** is that imported elements can only be correlated if they are imported in the same macrostep. In this context, two elements a, b are "correlated" in a run $\mathcal{A}_0 \vdash_\Pi \mathcal{A}_1 \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_i \vdash_\Pi \mathcal{A}_{i+1} \vdash_\Pi \ldots$, if there is an integer $j \in \mathbb{N}$ such that there exists an atom $R\overline{u}$ $\mathcal{A}_j \models R\overline{u}$, $\overline{u} = u_1 \ldots u_n$ is an n-tuple of elements from the domain of $\mathcal{A}_i$ and there are $k, l \in \{1, \ldots, n\}$ with $a = u_k$ and $b = u_l$. For a correlation of two imported elements (i.e., two elements that are in the reserve of the initial state) the respective import rules have to be nested, e.g. **import** u (**import** v $Ruv :=$ true **endimport**) **endimport**. The reasons are the following.

While importing an element the element can only be referenced inside the respective import rule. *After* having imported an element it can only be accessed via a forall rule as it can not become the content of a constant. Forall rules have to satisfy the guardedness condition. Therefore, only elements can be accessed which appear in a tuple $\overline{u}$ such that the atom $R\overline{u}$ is satisfied in the actual state. Therefore, if two imported elements have not been correlated before then they can not be correlated in the actual state. Applying this argument inductively and taking into account the above points, we obtain the note.

Definition 11.14. Let $\mathcal{A}$ be a state and $\overline{a} = a_1 \ldots a_n$ be a tuple of elements from the domain of $\mathcal{A}$.

The atomic type of $\overline{a}$ in $\mathcal{A}$ (in symbols $\mathrm{at}(\overline{a}, \mathcal{A})$) is the set of all atoms $R\overline{u}$ (where $\overline{u} = u_1 \ldots u_k$ and for $i \in \{1, \ldots, k\}$ u_i is an element the domain of $\mathcal{A}$ or one the variables $x_1, \ldots, x_n$) such that $\mathcal{A} \models R\overline{u}[x_1/a_1 \ldots x_n/a_n]$.

Definition 11.15. Let $\mathcal{A}$ be a state, $\overline{a}$ and $\overline{b}$ be tuples elements from the domain of $\mathcal{A}$.

$$\overline{a} \sim_{\mathcal{A}} \overline{b} \text{ if, and only if, } \mathrm{at}(\overline{a}, \mathcal{A}) = \mathrm{at}(\overline{b}, \mathcal{A})$$

Lemma 11.16. Let $\Pi \in \mathrm{GF}(\mathcal{DI})$ be an ASM, $\mathcal{A}$ be a state and A be the domain of $\mathcal{A}$. Furthermore, let $\mathcal{A} = \mathcal{A}_0 \vdash_\Pi \mathcal{A}_1 \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_i \vdash_\Pi \mathcal{A}_{i+1} \vdash_\Pi \ldots$ and $\overline{a}, \overline{b} \in A^n$ such that $\mathcal{A} \models \mathrm{reserve}(a_l)$ and $\mathcal{A} \models \mathrm{reserve}(b_l)$ for all $l \in \{1, \ldots, n\}$. Furthermore, let i be the smallest number such that $\mathcal{A}_\rangle \models \neg\mathrm{reserve}(a_l)$ for all $l \in \{1, \ldots, n\}$ and j be the smallest number $\mathcal{A}_j \models \neg\mathrm{reserve}(b_l)$ for all $l \in \{1, \ldots, n\}$. Let $k = \max(i, j)$.

If $\mathrm{at}(\overline{a}, \mathcal{A}_k) = \mathrm{at}(\overline{b}, \mathcal{A}_k)$ then $\mathrm{at}(\overline{a}, \mathcal{A}_m) = \mathrm{at}(\overline{b}, \mathcal{A}_m)$ for all $m \geq k$.

Theorem 11.17. There exists an algorithm computing for a slicing criterion S and an ASM $\Pi \in \mathrm{GF}(\mathcal{DI})$ a minimal S-slice of Π.

The proof for deterministic ASMs with **import** proceeds similar to the one for deterministic ASMs without import. Therefore, we do not give the complete proof but only the modifications and additional considerations that have to be carried when allowing the use of **import**.

As in the case without import we consider the runs on the standard states but here, we modify the received structure after every step such that we do not get the real runs but a kind of reduction of them. The reduction is explained more precisely in the following. The reason for the need of such a reduction is that the active domain of the states grows and therefore the argument that there are only finitely many states over the domain of the initial state does not work anymore.

After each execution step of the ASM, we count the number of effectively imported elements (with respect to S). If both numbers are equal then we continue with the next step. Else let d be the maximum of the nesting depth of import rules in the ASM and the number of inequalities appearing in the ASM. Let n be the number of different atomic types of d-tuples consisting of imported elements. Replace the state by the following one. The active domain consists of the active domain of the initial state and a number of additional elements which are described in the following. They are pairwise different and all of them are different from the elements in the active domain of the initial state. For every such atomic type t let n_t be the number of tuples $\overline{a}$ such that the atomic type of $\overline{a}$ in the new state is equal to t. If $n_t > d$ then there are exactly d tuples realizing t else there are exactly n_t tuples realizing t.

Note that starting from d, the exact number of tuples realizing an atomic type does not influence the number of (effectively) imported elements.

Again, as in the case of deterministic ASMs not using import, there is only a finite number of states that is reachable in this way from a given initial state, especially from a standard state. This number is computable from the initial state.

12 Variations in the Notion of Slicing

12.1 Dynamic and Static Slicing

In literature on slicing, there are two basic versions of slicing, namely static and dynamic slicing. For dynamic slicing of a program, a specific input is given and if we are only interested in a subprogram such that the behavior is the same for this specific input. In static slicing, the input is not given. We are interested in the same behavior for all possible inputs. Therefore, slicing in this work corresponds to static slicing.

In the case the classes $\mathcal{D}$ or $\mathcal{ND}$ (even if we allow function symbols to be in the vocabulary of an ASM in $\mathcal{D}$), dynamic slicing is rather easy. The computation graph of an ASM from $\mathcal{D}$ resp. $\mathcal{ND}$ is finite for a given initial state. Therefore, we can directly compare the computation graphs of two ASMs on a given state with respect to a slicing criterion. Consequently, comparing the computation graph an ASM Π on a given structure to the computation graphs of all elements of sub(Π) on this state leads to a minimal (dynamic) slice.

In the case of allowing import, it is not that easy but in the case where we have only relation and constant symbols, we can also compute a minimal (dynamic) slice. The algorithm is similar to the one in the case without import but we do not consider the complete computation graph. Namely, we do not insert *all* states into the computation but if the difference between two states is only caused by the import of different elements, then these states are identified and only one of them is inserted into the computation graph as node. Furthermore, we need a reduction similar to the one in the case of static slicing. Consequently, every node of the computation graph represents infinitely many states.

12.2 Limit the Number of Steps

The idea of limiting the number of steps is that possibly we are not interested in the complete runs of an ASMs but only in the initial segments of length $n \in \mathbb{N}$ of all states. This is for example the case if we know that an error occurs during the first n steps of a computation.

Similar to the notion of an S-slice, we can introduce the notion of an (S, n)-slice that is an element of sub(Π) which, with respect to S, behaves in the same

way as Π on every initial segment of length n.

Definition 12.1 (S-equivalence up to n). Let Π and Π' be ASMs, S be a slicing criterion and $n \in \mathbb{N}$.

1. Given a state $\mathcal{A}$ of Π resp. Π'. If for every initial segment of a run of Π $\mathcal{A} = \mathcal{A}_0 \vdash_\Pi \mathcal{A}_1 \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_i \vdash_\Pi \mathcal{A}_{i+1} \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_{n-1}$ there is an initial segment a run of Π' $\mathcal{A} = \mathcal{A}_0 \vdash_{\Pi'} \mathcal{A}'_1 \vdash_{\Pi'} \cdots \vdash_{\Pi'} \mathcal{A}'_i \vdash_{\Pi'} \mathcal{A}'_{i+1} \vdash_{\Pi'} \cdots \vdash_{\Pi'} \mathcal{A}_{n-1}$ such that $\mathcal{A}_i|_{L(\mathcal{A}_i,S)} = \mathcal{A}'_i|_{L(\mathcal{A}'_i,S)}$ for all $i \in \{0,\ldots,n-1\}$ then Π' (S,n)-captures Π on $\mathcal{A}$ ($\Pi \leq^{\mathcal{A}}_{S,n} \Pi'$).
2. Given a state $\mathcal{A}$ of Π resp. Π'. Π and Π' are called (S,n)-equivalent on $\mathcal{A}$ ($\Pi \equiv^{\mathcal{A}}_{S,n} \Pi'$) if $\Pi \leq^{\mathcal{A}}_{S,n} \Pi'$ and $\Pi' \leq^{\mathcal{A}}_{S,n} \Pi$.
3. If $\Pi \equiv^{\mathcal{A}}_{S,n} \Pi'$ for all states $\mathcal{A}$ of Π resp. Π' then Π and Π' are called (S,n)-equivalent ($\Pi \equiv_S \Pi'$).

Definition 12.2. Let Π be an ASM-program, S be a slicing criterion and $n \in \mathbb{N}$.
An (S,n)-slice of Π is an ASM-rule $\Pi_{S,n} \in \text{sub}(\Pi)$ such that $\Pi \equiv_{S,n} \Pi_{S,n}$.

For ASMs from $\text{GF}(\mathcal{D})$ and $\text{GF}(\mathcal{ND})$ and properties $\varphi \in \text{GF}$ we can prove the following theorem.

Theorem 12.3. There exists an algorithm computing for a slicing criterion S, $n \in \mathbb{N}$ and an ASM $\Pi \in \text{GF}(\mathcal{D})$ or $\Pi \in \text{GF}(\mathcal{D})$ a minimal (S,n)-slice of Π.

The proof of theorem 12.3 can be done completely analogously the proof of theorem 11.1 except that have to consider the runs on the standard states $\mathcal{A}_Q$ only up to the length n.

12.3 Conditioned Slicing

The idea of conditioned slicing is that we consider only states satisfying a given property. Formally, this can be formulated as follows.

Similar to the notion of an S-slice, we can introduce the notion of an (S,φ)-slice which is an element of $\text{sub}(\Pi)$ which, with respect to S, behaves in the same way as Π on every run whose initial state satisfies φ.

Definition 12.4. Let Π and Π' be ASMs, S be a slicing criterion and $\varphi \in \text{FO}$ be an FO-sentence.
If $\Pi \equiv^{\mathcal{A}}_S \Pi'$ for all states $\mathcal{A}$ of Π resp. Π' such that $\mathcal{A} \models \varphi$ then Π and Π' are called S-equivalent with respect to φ ($\Pi \equiv^{\varphi}_S \Pi'$).

Definition 12.5 (conditioned slice). Let Π be an ASM-program, S be a slicing criterion and $\varphi \in FO$ be an FO-sentence.
An (S,φ)-slice of Π is an ASM-rule $\Pi_{S,\varphi} \in \text{sub}(\Pi)$ such that $\Pi \equiv^{\varphi}_S \Pi_{S,\varphi}$.

For ASMs from GF($\mathcal{D}$) and GF($\mathcal{ND}$) and properties $\varphi \in$ GF we can prove the following theorem.

Theorem 12.6. There exists an algorithm computing for a slicing criterion S, $\varphi \in$ GF and an ASM $\Pi \in$ GF($\mathcal{D}$) or $\Pi \in$ GF($\mathcal{D}$) a minimal (S, φ)-slice of Π.

The proof of theorem 12.6 can be done completely analogously the proof of theorem 11.1 except that instead of T_{Π_1,Π_2} one has to use $T^{\varphi}_{\Pi_1,\Pi_2}$ where, for ASM Π_1 and Π_2, $T^{\varphi}_{\Pi_1,\Pi_2}$ is defined as $T_{\Pi_1,\Pi_2} \cup \{\varphi\}$. Instead of considering all $T^{\varphi}_{\Pi_1,\Pi_2}$-quasistates we only have to consider those $T^{\varphi}_{\Pi_1,\Pi_2}$-quasistates in which φ holds.

12.4 Semantic Slicing

Instead of giving a slicing as set of atoms, one could also have an ASM Π and a property φ (given some formalism resp. logic) such that one is interested in a minimal subprogram Π' of Π which behaves exactly the same way as Π with respect to φ.

Similar to the notion of an S-slice, we can introduce the notion of an φ-slice which is an element of sub(Π) that, with respect to the property φ, behaves in the same way as Π on every run.

Definition 12.7 (φ-equivalence). Let Π and Π' be ASMs and φ be a property given in some formalism.

1. Given a state $\mathcal{A}$ of Π resp. Π'. If for every run of Π $\mathcal{A} = \mathcal{A}_0 \vdash_\Pi \mathcal{A}_1 \vdash_\Pi \cdots \vdash_\Pi \mathcal{A}_i \vdash_\Pi \mathcal{A}_{i+1} \vdash_\Pi \ldots$ there is a run of Π' $\mathcal{A} = \mathcal{A}_0 \vdash_{\Pi'} \mathcal{A}'_1 \vdash_{\Pi'} \cdots \vdash_{\Pi'} \mathcal{A}'_i \vdash_{\Pi'} \mathcal{A}'_{i+1} \vdash_{\Pi'} \ldots$ such that, for all $i \in \mathbb{N}$, $\mathcal{A}_i \models \varphi$ if, and only if, $\mathcal{A}'_i \models \varphi$ then Π' φ-captures Π on $\mathcal{A}$ ($\Pi \leq^{\mathcal{A}}_{\varphi} \Pi'$).

2. Given a state $\mathcal{A}$ of Π resp. Π'. Π and Π' are called φ-equivalent on $\mathcal{A}$ ($\Pi \equiv^{\mathcal{A}}_{\varphi} \Pi'$) if $\Pi \leq^{\mathcal{A}}_{\varphi} \Pi'$ and $\Pi' \leq^{\mathcal{A}}_{\varphi} \Pi$.

3. If $\Pi \equiv^{\mathcal{A}}_{\varphi} \Pi'$ for all states $\mathcal{A}$ of Π resp. Π' then Π and Π' are called S-equivalent ($\Pi \equiv_{\varphi} \Pi'$).

Definition 12.8 (semantic slice). Let Π be an ASM-program and φ be a property given in some formalism.

An φ-slice of Π is an ASM-rule $\Pi_\varphi \in$ sub(Π) such that $\Pi \equiv_\varphi \Pi_\varphi$.

Theorem 12.9. There exists an algorithm computing for a formula $\varphi \in$ GF and an ASM $\Pi \in$ GF($\mathcal{D}$) a minimal φ-slice of Π.

The proof of this theorem can be done similar to the one of theorem 11.1 with the following changes. The slicing criterion S has to be replaced by the set of all subformulae of φ when constructing the quasistate. Furthermore, in the runs on the standard states we do not compare the contents of locations indicating an atom which is an instance of an element of the slicing criterion but check the property φ.

13 Equivalence of Abstract State Machines

In this chapter, we investigate the decidability of the equivalence of ASMs. Here, the equivalence of two ASMs requires that the runs of the ASMs on any initial state are the same.

Definition 13.1. Two (deterministic) ASM-programs Π_1 and Π_2 (over the same vocabulary) are equivalent if for all states $\mathcal{A}$ and $\mathcal{B}$ (of Π_1 and Π_2) the following holds:

$$\mathcal{A} \vdash_{\Pi_1} \mathcal{B} \text{ if, and only if, } \mathcal{A} \vdash_{\Pi_2} \mathcal{B}$$

In signs: $\Pi_1 \equiv \Pi_2$

Let $\mathcal{C}$ be a class of ASM programs. Then

$$\text{Equiv}(\mathcal{C}) := \{(\Pi_1, \Pi_2) : \Pi_1, \Pi_2 \in \mathcal{C} \text{ and } \Pi_1 \equiv \Pi_2\}$$

In general, the equivalence of two ASM programs is not decidable as the satisfiability problem for FO is not decidable and, for a formula α not containing Mode, **if** α **then** Mode := true **endif** is equivalent to Skip if, and only if, α is not satisfiable. Therefore, the decidability problem is strongly connected to the satisfiability problem for FO resp. the one for fragments of FO. But there are many (very large) classes of ASM programs for which equivalence is decidable.

In this chapter, we present a scheme of an algorithm that decides equivalence for two ASM-programs from one class (satisfying some properties given after the algorithm scheme). It is just a scheme as one has to insert an algorithm deciding satisfiability for a fragment of FO (this fragment results directly from the class $\mathcal{C}$ of ASMs).

13.1 The Absolute Guard

For future use, we define the absolute guard of a rule in a program. Essentially, the absolute guard of a rule is a list of all guards in whose scope the rule is situated.

Definition 13.2 (Absolute Guard). 1. For a list of items l and and item x, let $x \circ l$ be the list resulting by inserting x in the front of l.

2. Let Π be an ASM and R be a rule occurring only once in Π and $R \neq \text{Skip}$. The absolute guard of R in Π is defined (by induction on Π) as follows.
   - If Π is an update rule $s := t$ then the rule R must be equal to $s := t$ and $\text{absGuard}(\Pi, R) := ()$.
   - If Π is a parallel composition of the form **do-in-parallel** Π_1 Π_2 **enddo** then $\text{absGuard}(\Pi, R) := \text{absGuard}(\Pi_i, R)$ where R occurs in Π_i, $i \in \{1, 2\}$.
   - If Π is an if clause of the form **if** α **then** Π' **endif** then we define $\text{absGuard}(\Pi, R) := \alpha \circ \text{absGuard}(\Pi', R)$.
   - If Π is an import rule of the form **import** v Π' **endimport** then $\text{absGuard}(\Pi, R) := \text{absGuard}(\Pi', R)$.
   - If Π is a forall rule of the form **forall** $\overline{x} : \alpha(\overline{x})$ **do** Π' **endforall** then $\text{absGuard}(\Pi, R) := (\alpha(\overline{x}), \overline{x}) \circ \text{absGuard}(\Pi', R)$
   - If Π is a choice rule of the form **choose** $\overline{x} : \alpha(\overline{x})$ Π' **endchoose** then $\text{absGuard}(\Pi, R) := (\alpha(\overline{x}), \overline{x}) \circ \text{absGuard}(\Pi', R)$

The restriction that the rule R occurs only once in Π is not really a restriction as one can always easily construct an equivalent program in which every sub-rule occurs at most once.

W.l.o.g., we can assume that a variable bounded by **import** does not appear free in a guard or the left-hand side of an update rule. The reason is that every atom in which such a variable appears can be substituted by false (except that it appears on the left-hand side of an update-rule).

In the remainder this section, we investigate the information that is provided by an absolute guard. In order to simplify the formulation, we define two mappings conj and impl. Both map absolute guards to FO-formulae.

1. conj is defined by
   - $\text{conj}(()) := \text{true}$
   - if φ_0 is an FO-formula
     $$\text{conj}((\varphi_0, \varphi_1, \ldots, \varphi_{n-1})) := \varphi_0 \wedge \text{conj}((\varphi_1, \ldots, \varphi_{n-1}))$$
   - $\text{conj}(((\varphi_0', \overline{x})\varphi_1, \ldots, \varphi_{n-1})) := \exists \overline{x}(\varphi_0 \wedge \text{conj}((\varphi_1, \ldots, \varphi_{n-1})))$
2. impl is defined by
   - $\text{impl}() := \text{false}$
   - if φ_0 is an FO-formula then
     $$\text{impl}((\varphi_0, \varphi_1, \ldots, \varphi_{n-1})) := \varphi_0 \rightarrow \text{impl}((\varphi_1, \ldots, \varphi_{n-1}))$$

- $\text{impl}(((\varphi_0', \overline{x})\varphi_1, \ldots, \varphi_{n-1})) := \forall \overline{x}(\varphi_0 \rightarrow \text{impl}((\varphi_1, \ldots, \varphi_{n-1})))$

Essentially, we obtain the following information from an absolute guard.

Within one execution step of an ASM-program Π, the rule R appearing in Π is executed if, and only if, $\text{conj}(\text{absGuard}(\Pi, R))$ is satisfied. Therefore, if $\text{conj}(\text{absGuard}(\Pi, R))$ is not satisfiable then R is never executed. If we replace R by Skip in Π then we obtain a program which is equivalent to Π.

If, for any rule R appearing in the program Π, $\text{conj}(\text{absGuard}(\Pi, R))$ is valid then R is executed in every execution step of Π (independently from the state). Consequently, the execution of R within an execution step of Π does not depend on the valuation of the guards. If we replace R by Skip in Π (leading to $\Pi[R/\textbf{Skip}]$) and then construct **do-in-parallel** $\Pi[R/\textbf{Skip}]$ R **enddo** in case that $\text{free}(R) \neq \emptyset$ and else **do-in-parallel** $\Pi[R/\textbf{Skip}]$ **forall** $\overline{x}$: true **do** R **endforall enddo** (where $\text{free}(R) = \text{free}(\overline{x})$) then we obtain a program that is equivalent to Π.

Consider the rule **if** α **then** R **endif** (inside the program Π). If the formula $\text{impl}(\text{absGuard}(\Pi, R))$ is valid then we obtain a program equivalent to Π by replacing **if** α **then** R **endif** by R in Π.

Consider the rule **forall** $\overline{x}$: α **do** R **endforall** (in the program Π). If the formula $\text{impl}(\text{absGuard}(\Pi, R))$ is valid then we obtain a program equivalent to Π by replacing **forall** $\overline{x}$: α **do** R **endforall** by **forall** $\overline{x}$: true **do** R **endforall** in Π.

Let Π be a program and $s := t$ appear in Π. If $\text{impl}(\text{absGuard}(\Pi, s := t) \circ (s \leftrightarrow t))$ is valid then replacing the update-rule $s := t$ by **Skip** in Π leads to a program that is equivalent to Π.

13.2 Deciding the Equivalence of ASMs

In this section, we give a scheme for an algorithm for deciding equivalence of ASMs. It is a scheme in the sense that we have to substitute a satisfiability test for first-order formula at some place. Furthermore, this induces that this scheme does not provide an equivalence test for arbitrary ASMs as satisfiability is undecidable for first-order formula. But again, it provides an algorithm for certain classes of ASMs. We start by describing the idea for this scheme.

Consider the left-hand sides of the update rules with respect to unifiability. We can reduce the equivalence of ASMs to the equivalence of FO-formulae resulting from the guards. The content of a location $(Q, \overline{a})$ in a state $\mathcal{A}$ is changed by an ASM program Π if, and only if,

- Π is consistent in $\mathcal{A}$ and
- there exists an update rule $s := t$ such that s is unifiable with $Q\overline{a}$ (let σ be the corresponding substitution) and $\sigma(\text{conj}(\text{absGuard}(\Pi, s := t)^{\overline{x}}))$ is

satisfied
(where, for a absolute guard $p = (\varphi_0, \ldots, \varphi_{n-1})$, $p^{\overline{x}}$ results from p as follows:

- if $p = ()$ then $p^{\overline{x}} = p$
- if $p = \varphi_0 \circ (\varphi_1, \ldots, \varphi_{n-1})$ and φ_0 is an FO-formula then $p^{\overline{x}} = \varphi_0 \circ (\varphi_1, \ldots, \varphi_{n-1})^{\overline{x}}$
- if $p = (\varphi_0', \overline{y}) \circ (\varphi_1, \ldots, \varphi_{n-1})$ then $p^{\overline{x}} = (\varphi_0', \overline{y}') \circ (\varphi_1, \ldots, \varphi_{n-1})^{\overline{x}}$ where $\overline{y}'$ results from $\overline{y}$ by removing all variables occurring in $\overline{x}$

) and

- $\sigma(\text{impl}(\text{absGuard}(\Pi, s := t)^{\overline{x}} \circ (s \leftrightarrow t)))$ is not satisfied.

Two programs Π_1 and Π_2 are equivalent if, and only if, for all states $\mathcal{A}$ and all locations l in the content is changed by Π_1 if, and only if, it is changed by Π_2 (because a content which might be changed can only be true or false, one does not need to consider the way it is changed).

In order to formulate the algorithm scheme in a more compact way, we give some additional definitions in advance.

Definition 13.3. Let α be an atom, Π be an ASM-program (both over Υ) and $v \in \{\text{true}, \text{false}\}$. Then

$$[\alpha]^v_\Pi := \{\beta : \beta := v \text{ is an update rule in } \Pi \text{ and } \beta \text{ is unifiable with } \alpha\}$$

Definition 13.4. Let σ be a substitution, Π an ASM-program and $s := t$ an update rule occuring at most once in Π.
Then $\sigma(\Pi, s := t)$ is the program resulting from Π as follows.

1. Locate $s := t$ in Π
2. Replace $s := t$ by $\sigma(s) := \sigma(t)$
3. Replace all variables backwards starting from $s := t$ by $\sigma(x)$ until their bounding is reached; if they appear in a tuple $\overline{x}$ occuring in the form **forall** $\overline{x}$: $\alpha(\overline{x})$ and the variable x_j is replaced by the constant c then instead of $x_1 \ldots x_{j-1} c x_{j+1} \ldots x_n$ write $x_1 \ldots x_{j-1} x_{j+1} \ldots x_n$ (this does not cause new free variables in the rule as all occurrences of x_j are replaced by c.)

Definition 13.5. An import rule **import** v R **endimport** is effective in a program Π if none of the following points holds for R:

- R is equivalent to **Skip**
- for every update rule $s := t$ in R the one of the following holds:

  - v does not appear in s
  - v appears in s and t is equal to false
  - conj(absGuard($\Pi, s := t$)) is not satisfiable, v appears in s and t is not equal to false
  - the terms s and t are the same terms

- conj(absGuard(Π, R)) is not satisfiable.

eff(R) (which is introduced in the following definition) will be used in the algorithm only with non-effective import rules.

Definition 13.6. For a rule $R =$ **import** v R' **endimport**, let eff(R) be the subrule of R' satisfying the following:

- v does not appear (free) in eff(R)
- eff(R) does not contain update rules of the form $s := s$
- there is no rule $\tilde{R} \in \text{sub}(R)$ such that eff$(R, \Pi) \in \text{sub}(\tilde{R})$ and v does not appear in $\tilde{R}$ and $\tilde{R}$ does not contain an update rule of the form $s := s$.

Note that eff(R) is unique for an ASM-rule R.

In the remainder of this chapter, we give an algorithm (scheme) deciding the equivalence of two ASMs satisfying certain conditions. The precise conditions are given after the algorithm. For example, all ASMs from GF($\mathcal{D}$) satisfy these conditions. Therefore, the following is an algorithm deciding for two ASMs from GF($\mathcal{D}$) whether they are equivalent.

Input. An input of the algorithm (scheme) are two ASMs Π_1, Π_2 that are both from a class ASM-programs satisfying the conditions given below the algorithm (e.g. from GF($\mathcal{D}$)).

Algorithm. The algorithm scheme proceeds in the following nine steps.

1. found := false
2. Replace in Π_1, Π_2 every update rule of the form $s := t$ where s and t are boolean-valued terms (and $t \notin \{\text{true}, \text{false}\}$) by

do-in-parallel
  if t **then** $s :=$ true **endif**
  if $\neg t$ **then** $s :=$ false **endif**
enddo

3. for $i \in \{1,2\}$, let $\tilde{\Pi_1}$ be the program resulting from Π_i by replacing all non-effective import-rules R in Π_1 with $\text{eff}(R)$.

4. $\sharp_1 :=$ number of import rules in $\tilde{\Pi_1}$

5. $\sharp_2 :=$ number of import rules in $\tilde{\Pi_2}$

6. if $\sharp_1 \neq \sharp_2$ then reject; end
   /*
   The reason for the rejection at this point is the following. If $\sharp_1 > \sharp_2$ then there exist structures $\mathcal{A}, \mathcal{B}_1, \mathcal{B}_2$ such that $\mathcal{A} \vdash_{\Pi_1} \mathcal{B}_1$, $\mathcal{A} \vdash_{\Pi_2} \mathcal{B}_2$ and $|\text{ad}(\mathcal{B}_1)| > |\text{ad}(\mathcal{B}_2)|$ and therefore $\mathcal{A} \not\vdash_{\Pi_1} \mathcal{B}_2$ as the successor of $\mathcal{A}$ with respect to Π_1 is unique up to import isomorphism.
   */

7. // This step reflects the main part of the algorithm.

   forall bijective mappings $f : \{1, \ldots, \sharp_2\} \to \{1, \ldots, \sharp_1\}$
   //note that at this point $\sharp_1 = \sharp_2$ and therefore $f : \{1, \ldots, \sharp_1\} \to \{1, \ldots, \sharp_1\}$
   for $i \in \{1,2\}$
   $\Pi_i^f :=$ program resulting from $\tilde{\Pi_i}$ by replacing all occurences of the variable bounded by the j-th import by

$$\begin{array}{lll} c_{f(j)} & \text{if} & i = 1 \\ c_j & \text{if} & i = 2 \end{array}$$

   and removing all appearances of **import** and **endimport**
   forall $v \in \{\text{true}, \text{false}\}$
   forall $i, j \in \{1,2\}$, $j \neq i$
   forall update rules $\alpha(\overline{x}) := v$ in Π_i
   forall $L \in \mathcal{P}([\alpha]^v_{\Pi_i})$
   // for a set X, $\mathcal{P}(X)$ denotes the potential set of X
   $\sigma := \text{mgu}(L)$
   if $[\bigvee_{t \in L} \text{conj}(\text{absGuard}(\sigma(\Pi_i^f, t := v), \sigma(t) := v))] \leftrightarrow$

$$\begin{aligned} &[(\bigvee_{\substack{t \in [\sigma(\alpha)]^v_{\Pi_j^f} \\ \tilde{\sigma} = mgu(t, \sigma(\alpha))}} \text{conj}(\text{absGuard}(\tilde{\sigma}(\Pi_j^f, t := v), \tilde{\sigma}(t) := v))) \\ &\vee (\bigvee_{[\sigma(\alpha)]^v_{\Pi_j^f} = \emptyset} \sigma(t))] \\ &\wedge \bigwedge_{\substack{k,l \in \{1,\ldots,\sharp_1\} \\ k \neq l}} c_k \neq c_l \end{aligned}$$

   is not valid
   then continue with the next mapping if there is one
   end // forall

```
        end // forall
      end // forall
      found := true
      // the actual mapping witnesses the equivalence
    end // for
  end // forall
```

8. if ¬found then reject; end
   /*
   no mapping witnesses the equivalence and therefore, the ASMs are not equivalent.
   /*

9. accept; stop
   /*
   if this step is reached then all validity tests have been carried out. If found is true then validity is received for at least one mapping. Therefore, a location is updated positively resp. negatively resp. not updated (with respect to import) in an arbitrary state by Π_1 iff it is updated positively resp. negatively resp. not updated in this state by Π_2. Therefore, Π_1 and Π_2 are equivalent.

   The other direction is obvious. In the case that they are not equivalent, there has to be a difference in the two programs.
   */

Let us consider the (pre)conditions that are necessary to obtain an algorithm from this scheme. The steps 1, 2, 3, 4, 5, 6 can be carried out without any assumptions on the considered class of ASM programs.
The only problem are the validity tests in steps 7. Let $\mathcal{C}$ be a class of ASM-programs and

$$\mathcal{F}_\mathcal{C} := \{\varphi |\ \varphi \text{ is an FO-formula and there exist ASM-programs } \Pi_1, \Pi_2 \in \mathcal{C} \text{ such that } \varphi \text{ is tested for validity during the execution of the algorithm on } \Pi_1, \Pi_2\}.$$

In order to obtain a concrete algorithm from the above scheme, there has to exist a fragment $\mathcal{L}$ of FO such that $\mathcal{F}_\mathcal{C} \subseteq \mathcal{L}$ and the validity problem is decidable for $\mathcal{L}$.

E.g., for $\mathcal{L}$, choose the guarded fragment of first-order logic (with equality) and for $\mathcal{C}$ choose GF($\mathcal{D}$). In this case, $\mathcal{F}_\mathcal{C} \subseteq$ GF. It is already known that the satisfiability problem for GF is decidable. As GF is closed under negation, this

is also true for the validity problem. Therefore, the algorithm works for the class of clique-guarded ASM programs.

There are many more examples for such classes of ASM programs.

Complexity

Having proven the decidability of equivalence for some class of ASMs by giving an algorithm, we are also interested in its complexity.

Let $T(n)$ be the function giving the complexity for the validity problem of $\mathcal{F}_{\mathcal{C}}$.

The length of the formulae that are checked for validity is bounded by the size of the program. The number of validity checks that are carried out is bounded for one mapping by $O(n \cdot 2^n)$ (where n is the size of the input). Therefore, the complexity for checking equivalence is in $2^{O(n \cdot \log n)} \cdot T(n)$ where n is the sum of the sizes of the programs.

As described above, the satisfiability problem for FO and fragments of FO can be easily (in linear time) reduced to the equivalence problem for ASM programs. This yields the respective hardness results if the fragments are already proven to be hard for some complexity class (above linear time). E.g., the satisfiability problem for GF is 2EXPTIME-complete. Therefore, the equivalence problem for ASMs from GF($\mathcal{D}$) is 2EXPTIME hard. Together with the above calculation, equivalence for GF($\mathcal{D}$) is 2EXPTIME-complete.

Part IV

Abstract State Machines Capture Quantum Algorithms

14 The ASM Thesis for Sequential and Parallel Algorithms

From now on, we do not consider ASMs merely as a specification formalism but as a formal computation model. As a consequence, we are confronted with different kinds of questions. One such question results from the ASM thesis saying that *"every algorithm of any kind is modeled, step by step and on its natural abstraction level, by an abstract state machine"*. Though this thesis is broadly accepted, there does not exist a proof for the general case. But in two cases (namely sequential algorithms and parallel ones) the ASM thesis has been derived from certain basic postulates (see [5, 21]). Here, we do the same for quantum algorithms.

We first present a number of basic postulates for quantum algorithms. We then prove that all objects satisfying these postulates can be modeled step by step and on its natural abstraction level by an ASM. We prove this by concretely constructing a simulating ASMs for an arbitrary quantum algorithm.

In order to obtain a better understanding of this approach we shortly resume the postulates for sequential and parallel algorithms. For sequential algorithms we follow here the presentation of [5] (rather than [21]) which makes the output that the algorithm produces at each step explicit, in form of a function out_A.

Postulates for sequential algorithms. A sequential algorithm with output is an object A satisfying the following three postulates.

S1: Sequential Time. A is associated with a set $\mathcal{S}(A)$ of *states*, a subset $\mathcal{I}(A) \subseteq \mathcal{S}(A)$ of *initial states*, a function $\tau_A : \mathcal{S}(A) \to \mathcal{S}(A)$, called the *one-step transformation* of A, and a function out_A assigning to each state $X \in \mathcal{S}(A)$ a set (or multiset) of elements of X, called the output of A at state X.

S2: Abstract State. All states of A are first-order structures with the same finite vocabulary and the same universe. Both $\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphisms, and any isomorphism between two states X and Y is also an isomorphism between $\tau_A(X)$ and $\tau_A(Y)$, and maps $\text{out}_A(X)$ to $\text{out}_A(Y)$.

S3: Uniformly Bounded Exploration. There is a finite set T of terms in the vocabulary of A such that, whenever two states X and Y coincide on T, then the update sets and outputs of A on X and Y are the same: $\Delta(A, X) = \Delta(A, Y)$ and $\text{out}_A(X) = \text{out}_A(Y)$.

We recall that the update set $\Delta(A, X)$ of A on state X represents the atomic changes in X produced by the algorithm A. It can be defined as the set of expressions $f(\overline{a}) = b$ that are true in $\tau_A(X)$ but not in X (where f is a function symbol, and $\overline{a}$ and b are elements of the universe).

A sequential ASM is an ASM that uses only

- function updates: $f(t_1, \ldots, t_m) := t_0$,
- parallel composition of two rules: **do-in-parallel** R_1, R_2 **enddo**,
- if rules: **if** φ **then** R_1 **else** R_2 **endif** where φ is a Boolean term (i.e. a formula without quantifiers), and
- output rules: **output** t, where t is any term.

The main result of [21] is that every sequential algorithm satisfying postulates (S1) - (S3), can be simulated, step by step, by a sequential ASM. Step by step simulation means that the two algorithms have the same states and initial states, the same one-step transformation, and the same output function.

Parallel algorithms and parallel ASMs. The notion of a parallel algorithm is meant here to capture algorithms that do still work in discrete time steps, governed by a global clock (as opposed to distributed, asynchronous algorithms), but with unbounded parallelism in each time step. Hence, parallel algorithms need not satisfy the Uniformly Bounded Exploration Postulate (S3). We will see that quantum algorithms are, in this sense, parallel algorithms.

For the explanation of the postulates for parallel algorithms we limit ourselves to the aspects that we need for dealing with quantum algorithms. A parallel algorithm splits into smaller processes. Ultimately this splitting leads to atomic processes (called *proclets*), each of which executes the same (sequential!) algorithm. In the simplest case, the global update set of the algorithm is just the union of the 'local' updates produced independently by the proclets. In general however, parallel algorithms need to combine and integrate the updates produced by their proclets to a global update set in a more complicated way that involves communication between proclets.

Blass and Gurevich argue that this communication of proclets and the manipulation of messages and update sets is best described in terms of *multisets and multiset operations*. The difference of multisets from sets is that multisets may have multiple occurrences of the same element. Formally, a multiset M over A

is given by a function $\mathrm{Mult}_M : A \to \mathbb{N}$ giving for each $a \in A$ the multiplicity of $a \in M$ (only finite multiplicities are used here). We use the notation $\{\!\{\ldots\}\!\}$ to denote multisets. In particular, for a term $t(x)$, a set or multiset r, and a formula $\varphi(x)$, we write $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ to denote the multiset M of all values of $t(x)$ subject to the condition that x is in r and satisfies φ; more formally $\mathrm{Mult}_M(a) = \sum_{x:\varphi(x) \text{ and } t(x)=a} \mathrm{Mult}_r(x)$ is the multiplicity of a in the multiset M.

The **Background Postulate** for parallel algorithms states that the abstract states of parallel algorithms contain (among other things) all finite multisets of elements of the state and some basic operations on them. This is reflected in the definition of parallel ASMs which extend sequential ASMs in two essential ways:

Comprehension terms: Given terms $t(x), r$, and $\varphi(x)$ one may build the comprehension terms $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$.

Unbounded parallelism: If x is a variable, $R(x)$ is a rule and r is a term without free occurrences of x, then **forall** $x \in r$ **do** $R(x)$ **enddo** is a rule.

Note that both operations require parallelism (they do not satisfy the Uniformly Bounded Exploration Postulate). Also for parallel algorithms, the ASM thesis can be derived from appropriate postulates [5].

Outline of Part IV. The remainder of this part consists of five chapters. First, we give a brief introduction to quantum algorithms. Based on this, we spend some thoughts on the simulation of quantum algorithms by ASMs. In order to obtain a better understanding of the connection between quantum algorithms and ASMs, we consider some examples of quantum computation models respectively quantum algorithms and give a simulation by an ASM for each of these examples. We then present basic postulates characterizing precisely the class of quantum algorithms. In the last chapter, we prove the ASM thesis for quantum algorithms by use of the postulates.

15 Models of Quantum Computation

For detailed background on quantum computing we refer to the textbooks [18, 23, 8]. There are several theoretical models of quantum algorithms, including quantum circuits, quantum Turing machines, quantum automata, and quantum programming. What all these models have in common is that a *state* of a quantum algorithm is a *superposition of states* of a corresponding classical model.

Mathematically, a state of a quantum algorithm is a unit-length vector $|\psi\rangle \in H$ of a Hilbert space.

Definition 15.1. A Hilbert space is a complex vector space H, equipped with an inner product mapping pairs of vectors $|x\rangle, |y\rangle$ to complex numbers $\langle x|y\rangle$. Further H must be complete with respect to the norm $\|x\| := \sqrt{\langle x|x\rangle}$ induced by the inner product (i.e. every Cauchy sequence in H has a limit in H).

Quantum bits and quantum registers. A quantum bit (shortly, qubit) is a vector in the two-dimensional Hilbert-space H_2 spanned by the orthonormal vectors $|0\rangle$ and $|1\rangle$. Hence, a qubit has the form $\alpha|0\rangle + \beta|1\rangle$ with $\|\alpha\|^2 + \|\beta\|^2 = 1$. If we *measure* such a qubit, the probability to observe 0 is $\|\alpha\|^2$ and the probability to observe 1 is $\|\beta\|^2$. In the case of an n-qubit quantum register, we work in the 2^n-dimensional Hilbert-space $H_{2^n} = H_2 \otimes \cdots \otimes H_2$ with orthonormal basis $\{|w\rangle : w \in \{0,1\}^n\}$ which we always assume to be ordered lexicographically, with $|00\cdots0\rangle$ as first bases vector and $|11\cdots1\rangle$ as the last one (the order of a basis is important for the representation of linear transformations by matrices). A state of such a quantum register has the form $\sum_{w\in\{0,1\}^n} \alpha_w|w\rangle$ with $\sum_{w\in\{0,1\}^n} \|\alpha_w\|^2 = 1$.

A state $|\psi\rangle$ in the product space $H \otimes H'$ is decomposable if it can be written as a tensor product $|\psi\rangle = |\varphi\rangle \otimes |\varphi'\rangle$ with $|\varphi\rangle \in H$ and $|\varphi'\rangle \in H'$. Otherwise it is called *entangled*.

Quantum circuits. The most influential model for quantum computation (probably due to the fact that most of the work on quantum algorithms is done by physicists) has been the quantum circuits model. There is not a uniform terminology here. While some authors use the notions quantum circuits, quantum algorithm and quantum computers interchangeably, others reserve the term quantum circuit for a sequence of quantum gates (with or without measurement), and

view quantum computers or quantum algorithms as a more general model that may combine classical parts with quantum circuits and measurement steps. We follow here the second approach.

Definition 15.2. A matrix U is unitary, if its transpose conjugate U^* is its inverse: $U^*U = I$.

Definition 15.3. A quantum gate on m qubits is a unitary transformation U on a 2^m-dimensional Hilbert space H_{2^m}. Such a transformation can be described by a unitary matrix which we also denote by U. For our purposes, it is convenient to view this matrix as a function $U : \{0,1\}^m \times \{0,1\}^m \to \mathbb{C}$. The entry $U(x,y)$ gives the probability amplitude of the transition from base state $|y\rangle$ to base state $|x\rangle$.

Example. The Hadamard gate operates on H_2 and is defined by the matrix

$$H = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

which maps $|0\rangle$ to $|0'\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ to $|1'\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

Example. The CNOT (controlled NOT) gate is another gate which is used rather often. It operates on $H_4 = H_2 \otimes H_2$ and can be described by the operation $|a,b\rangle \to |a, a \oplus b\rangle$ or, equivalently, by the matrix

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Definition 15.4. Let Ω be a fixed collection of quantum gates. A *quantum circuit* on n-qubits over Ω is a unitary transformation on H_{2^n} which is defined by a finite sequence of operations of the form

$$\text{“apply } U \text{ to the qubits } i_1, \ldots, i_m\text{”} \tag{15.1}$$

where U is an m-qubit gate from Ω, and $i_1, \ldots, i_m$ are distinct indices in the set $\{1, \ldots, n\}$.

Mathematically, operation 15.1 is described by the matrix

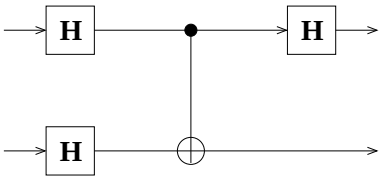
$$U_{i_1 \ldots i_m} := P^{-1}_{i_1 \ldots i_m} (U \otimes I_{2^{n-m}}) P_{i_1 \ldots i_m}$$

where $P_{i_1 \ldots i_m}$ is a permutation matrix moving the qubits $i_1, \ldots, i_m$ to the positions $1, \ldots, m$ and $(U \otimes I_{2^{n-m}})$ is the tensor product of U (operating on the first m

qubits) with the identity matrix of dimension 2^{n-m} (operating on the remaining qubits). Given the $2^m \times 2^m$-matrix U, and the numbers $i_1, \ldots, i_m$ the entries of the $2^n \times 2^n$-matrix $U_{i_1 \ldots i_m}$ are

$$U_{i_1 \ldots i_m}(x, y) := \begin{cases} U(x_{i_1} \cdots x_{i_m}, y_{i_1} \cdots y_{i_m}) & \text{if } x_k = y_k \text{ for all } k \notin \{i_1, \ldots, i_m\} \\ 0 & \text{otherwise.} \end{cases}$$

Remark. A popular notation for quantum circuits represents each qubit by a wire and the gates operating on them by boxes or circles (with special symbols for popular gates like Hadamard or CNOT). Here is an example.



The meaning of this picture is that before the application of the CNOT gate Hadamard gates are applied to both qubits separately. Afterwards, a Hadamard gate is applied to the first qubit. Mathematically, the circuit is described by the matrix $(H \otimes I_2)C(H \otimes H)$. Consider its operation on $|01\rangle$. The application of the Hadamard-gates leads to $\frac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle)$. The application of the CNOT-gate then leads to $\frac{1}{2}(|00\rangle + |11\rangle - |01\rangle - |10\rangle)$. The application of the Hadamard-gate to the first qubit finally leads to $\frac{1}{\sqrt{2}}(|10\rangle - |11\rangle)$.

Measurement. Let $H = M_1 + \cdots + M_k$ be a decomposition of the Hilbert space H into orthogonal subspaces. Then every $|\psi\rangle \in H$ can be written in a unique way as $|\psi\rangle = |\psi_1\rangle + \cdots + |\psi_k\rangle$, with $|\psi_i\rangle \in M_i$. Measurement of a state $|\psi\rangle$ (of unit length) with respect to the *observable* $\{M_1, \ldots, M_k\}$ means that $|\psi\rangle$ is mapped to $\beta|\psi_i\rangle$ for one i, where i is picked with probability $\langle\psi_i|\psi_i\rangle$ and $\beta|\psi_i\rangle$ has unit length.

While one can in principle do measurements with respect to any orthogonal decomposition of the state space, in practice it normally suffices to do measurements with respect to one or more qubits in the computational basis. More precisely, we need operations

"measure with respect to qubits $i_1, \ldots, i_m$"

which means that the current state $|\psi\rangle = \sum_{x \in \{0,1\}^n} \alpha(x)|x\rangle$ is written as $|\psi\rangle = \sum_{u \in \{0,1\}^m} |\psi_u\rangle$ where $|\psi_u\rangle = \sum_{x : x_{i_1} \cdots x_{i_m} = u_1 \cdots u_m} \alpha(x)|x\rangle$. We pick one u according to the probability distribution $p(u) = \sum_{x : x_{i_1} \cdots x_{i_m} = u_1 \cdots u_m} \|\alpha(x)\|^2$, and project $|\psi\rangle$ to $\beta|\psi_u\rangle$ with $\beta = 1/\sqrt{p(u)}$ so that the resulting vector has again unit length.

We view a quantum algorithm as a combination of a classical sequential algorithm with quantum circuits. In its classical phases the algorithm may prepare any computational basis state $|x\rangle$ (where $x \in \{0,1\}^n$ for arbitrary n) and in its quantum phases a quantum circuit is applied. To summarize, a general model of quantum algorithms can be described as follows.

Classical and quantum part. A quantum algorithm is a combination of a classical algorithm with quantum circuits. The collection Ω of quantum gates that are used by the algorithm is fixed, but the quantum circuits used by the algorithm may work with any number n of qubits.

Quantum mode. In quantum mode, a quantum circuit on some number n of qubits is applied to a computational basis state $|x\rangle = |x_0 \dots x_{n-1}\rangle$. As result the circuit produces a (usually entangled) state $|\psi\rangle \in H_{2^n}$.

Measurement. After application of a quantum circuit, the algorithm returns to classical mode (or another quantum circuit) by a measurement step in the computational basis of one or more qubits.

16 Simulating Quantum Algorithms by ASMs

16.1 General Considerations

An obvious, but minor, obstacle for the simulation of quantum algorithms by ASMs is that quantum algorithms are genuinely *probabilistic* algorithms, genuine in the sense that the probabilism is not simulated by a deterministic generation of pseudo-random numbers, but by the inherent probabilism in quantum mechanical processes. We thus have to assume that the simulating ASMs have access to true random numbers. Therefore, our quantum ASM will make use of a dynamic function random providing at each state access to a random real number in the interval $[0, 1]$.

The power of quantum computing comes from the possibility to process in one step a large number of entangled classical states. To put it differently, quantum algorithms go beyond classical concepts of sequential algorithm.

Lemma 16.1. Quantum algorithms are not sequential algorithms in the sense of postulates **(S1) - (S3)** and can thus not be simulated by sequential ASM.

Proof. Both unitary evolution steps and measurement steps violate the Uniformly Bounded Exploration Postulate. □

On the other side, it is not difficult to simulate quantum algorithms by parallel ASMs in the sense of [5] (we will make this precise below). But this is not the point. The point of quantum computing is to use the inherent parallelism in quantum effects for constructing more efficient algorithms than classical sequential algorithms. Indeed quantum algorithm can perform certain tasks (like searching databases or factoring integers) more efficiently than classical sequential algorithms can do. But we do not want to add anything else than "being quantum" to the paradigm of sequential algorithms.

A quantum state is a superposition of classical states. A computation step of a quantum algorithm changes a quantum state by modifying in parallel the underlying classical states (the base vectors of the Hilbert space). Hence it is clear that everything a quantum algorithm can do can also be done by a classical parallel algorithm. But quantum computers (if they can be built) implement exponential parallelism without using exponential resources!

From the quantum point of view, quantum algorithms are sequential, the only parallelism being inside the basic operations of unitary transformation and measurement. An adequate simulation of quantum algorithms by ASMs should take this into account. Rather than going up to arbitrary parallel ASMs, we therefore define quantum ASMs as sequential ASMs, enriched by two kinds of rules that take care of the quantum operations.

States. Quantum ASMs must have states that are sufficiently rich to describe the mathematics of quantum gates and quantum measurement. For circuits over a fixed basis Ω of quantum gates, the background of the states should contain, along with the requirements given in [5], the basic arithmetic operations on complex numbers, natural numbers, binary strings, appropriate descriptions for the quantum gates in Ω (appropriately parameterized if Ω is infinite), and the like. In particular we assume the presence of basic operations on multisets, and a summation operator mapping any finite multiset M of complex numbers to the sum over its elements. (Formally, if $M \subseteq \mathbb{C}$ is given by $\mathrm{Mult}_M : \mathbb{C} \to \mathbb{N}$, then $\sum M = \sum_{z\in\mathbb{C}} \mathrm{Mult}_M(z) \cdot z$.)

Suppose the algorithm enters a quantum phase on n qubits, starting with a classically prepared vector $|x\rangle = |x_0 \dots x_{n-1}\rangle$ in the computational basis of H_{2^n}, and should perform a sequence of unitary operations of the type described in Definition 15.4. The corresponding state X of the algorithm must contain the information n and $|x\rangle$. The entangled states $|\psi\rangle$ that will be produced by the unitary evolution of the system can be described by a dynamic function $\Psi : \{0,1\}^* \to \mathbb{C}$ where

$$|\psi\rangle = \sum_{x\in\{0,1\}^n} \Psi(x)|x\rangle.$$

Given n and x the ASM initializes Ψ by $\Psi(x) = 1$ and $\Psi(y) = 0$ for $y \neq x$.

Unitary transformations. To make precise how the operation

$$\text{“apply } U \text{ to the qubits } i_1, \dots, i_m\text{”}$$

is described by an ASM rule observe that for every basis state $|x\rangle$ the relevant (i.e. non-zero) entries $U_{i_1\dots i_m}(x, y)$ of the matrix $U_{i_1\dots i_m}$ are those for which $x_k = y_k$ for all $k \notin \{i_1, \dots, i_m\}$. For $x \in \{0,1\}^n$, $i < n$ and $z \in \{0,1\}$, let $x[i/z]$ be the word obtained from x by replacing the i-th bit x_i by z. The application of $U_{i_1\dots i_m}$ to the state defined by $\Psi : \{0,1\}^n \to \mathbb{C}$ is described by the rule

forall $x \in \{0,1\}^n$ **do**

$$\Psi(x) := \sum \{\!\{ U(x_{i_1} \cdots x_{i_m}, z_1 \cdots z_m)\Psi(y) : z_1, \dots, z_m \in \{0,1\}, y = x[i_1/z_1] \cdots [i_m/z_m] \}\!\}$$

enddo

Measurement. Given an entangled state $|\psi\rangle \in H_{2^n}$ and indices $i_1, \ldots, i_m$, measurement with respect to qubits $i_1, \ldots, i_m$ means computing the probabilities $p(u) = \langle\psi_u|\psi_u\rangle$ of the associated decomposition $|\psi\rangle = \sum_{u\in\{0,1\}^m} |\psi_u\rangle$, selecting one u according to this distribution, and projecting $|\psi\rangle$ to $\beta|\psi_u\rangle$.

We can implement this by selecting a random number $r \in [0,1]$, decomposing the unit interval into segments $S(u)$ (where $u \in \{0,1\}^m$) of length $p(u)$ and picking the u for which r is contained in $S(u)$. Again, this can be described in ASM-notation by a straightforward application of a **forall-do** rule and summation over a multiset. We assume that random is a nullary function whose value in each state is a random real number between 0 and 1. (At each fixed state the value of random of course is the same when it is called several times in parallel.)

forall $x \in \{0,1\}^n, u \in \{0,1\}^m$ **do**

if $\varphi(x,u)$ **then** $\Psi(x) := 1/\sqrt{p(u)}\Psi(x)$ **else** $\Psi(x) := 0$

enddo

where

$$\varphi(x,u) := (\ell(u) \leq \text{random} < \ell(u) + p(u)) \wedge x_{i_1} = u_1 \wedge \cdots \wedge x_{i_m} = u_m$$
$$\ell(u) := \sum\{\!\{p(v) : v \in \{0,1\}^m : v <_{\text{lex}} u\}\!\}$$
$$p(u) := \sum\{\!\{\|\Psi(y)\|^2 : y \in \{0,1\}^n : y_{i_1} = u_1 \wedge \cdots \wedge y_{i_m} = u_m\}\!\}$$

16.2 Examples

After we have spent some general thoughts on simulating quantum algorithms by ASMs, we present four concrete examples. We consider two basic quantum computation models (namely, quantum Turing machines and quantum circuits) and two of the most common quantum algorithms (namely Shor's factorization algorithm and Grover's search algorithm). In each case, we give a simulation by ASMs.
For further details on the computation models and the algorithms, see e.g. [18, 23].

16.2.1 Quantum Turing Machines

A *quantum Turing machine* (QTM) is a tuple $M = (Q, \Sigma, q_0, q_a, q_r, \delta)$ with alphabet Σ, finite set of control states Q, initial state q_0, accepting state q_a, rejecting

state q_r, and a transition function $\delta : Q \times \Sigma \times \Sigma \times Q \times \{-1, 0, 1\} \to \mathbb{C}$, where $\delta(q, \sigma, , q', \sigma', m)$ defines the amplitude that whenever the machine is in state q and reads symbol σ, it replaces σ with σ', enters state q' and moves the head in direction m. The transition function must satisfy the condition that the induced transformation on the configuration space is unitary.

As in the case of classical Turing machines, a configuration of a QTM is determined by the content of the tape, the current state, and the position of the head. Let C_M denote the set of all configurations of M and let H_M be the Hilbert space with orthonormal basis $\{|c\rangle : c \in C_M\}$. The transition function δ induces a mapping $a : C_M \times C_M \to \mathbb{C}$ giving for every pair (c, c') of configurations the amplitude of the transition of M from the basis state $|c'\rangle$ to $|c\rangle$. The *quantum evolution* defined by M on H_M maps $|\psi\rangle = \sum_{c \in C_M} \Psi(c)|c\rangle$ to $U_M|\psi\rangle = \sum_{c \in C_M} \sum_{c' \in C_M} a(c, c')\Psi(c')|c\rangle$. The standard measurement is the measurement with respect to the basis $\{|c\rangle : c \in C_M\}$ (at the end of the computation).

Simulating Quantum Turing Machines.

Recall that the states of an algorithm are supposed to contain everything that is necessary to determine the future progress of a computation. For a QTM M this means that the states contain the static function basis whose value is the set of configurations C_M. Further the states contain functions state: $C_M \to Q$, content: $C_M \times \mathbb{Z} \to \Sigma$, head: $C_M \to \mathbb{Z}$ and a relation replace $\subseteq C_M \times \Sigma \times Q \times \{-1, 0, 1\} \times C_M$ describing the configurations and the cross-linking between the configurations (namely, replace(c, σ, q, z, c') holds if, and only if, c' results from c by replacing the contents of the head(c)-th cell of c with σ, the state of c with q and moving the head according to z).

In the remainder on this section. we formally describe a simulation. Given a quantum Turing machine $M = (\Sigma, Q, q_0, q_f, \delta)$ with

$$\delta : Q \times \Sigma \times \Sigma \times Q \times \{-1, 0, +1\} \to \mathbb{C}_{[0,1]}$$

We present the three components of a simulating ASM, namely the background of the ASM, the initialization mapping and the ASM program.

Background. The background of the simulating ASM must contain at least

- $(\mathbb{Z}, \text{succ})$, integers with the usual successor function,
- $(\mathbb{C}, +, \cdot)$, the complex numbers with the usual addition and multiplication,
- a mapping Σ assigning to every finite multiset of complex numbers the sum of its members respectively multiplied with their multiplicity within the multiset,

- sets corresponding to $\Sigma \cup \{\square\}, Q$, and
- an infinite countable set C_M together with the functions

$$\begin{aligned} \text{state} &: C_M \to Q \\ \text{inscr} &: C_M \times \mathbb{Z} \to \Sigma \cup \{\square\} \\ \text{head} &: C_M \to \mathbb{Z} \end{aligned}$$

with the following intended meanings.

  - state(c) is the state of the configuration of c
  - inscr(c, i) is the i-th symbol in c if i is $\leq$ the word in c, $\square$ else
  - head is the head position in the configuration c

Initial states and initialization mappings. A initial state must contain a configuration $c_0 \in C_M$ with $S(c_0) = 1$ and for $c \neq c_0$ $S(c) = 0$. Furthermore, it is required that state(c_0) = q_0 and head(c_0) = 1.

Every possible input $w \in \Sigma^*, w = w_1 \dots w_n$ of the Turing machine (and therefore, for the ASM) can be mapped to an initial state by choosing in the above definition for the configuration $q_0 w$ for C_0 with

$$\begin{array}{lcll} \text{inscr}(c_0, i) & = & w_i & \text{if } i \in \{1, \dots, n\} \\ \text{inscr}(c_0, i) & = & \square & \text{else} \end{array}$$

The ASM program. The work of the above quantum Turing machine is simulated by the ASM program

forall $c : c \in C_M$ **do**
  $S(c) := \sum \{\!\{ S(c') \cdot a(c', c) : c' \in C_M : S(c') \neq 0 \}\!\}$
endforall

where $a(c_1, c_2)$ is the amplitude for a transition from c_1 to c_2. It be calculated from δ as follows.

If head(c_2)$-$head(c_1) $\in \{-1, 0, 1\}$ and inscr(c_1, i) = inscr(c_2, i)$\forall i \neq$ head(c_1) then $a(c_1, c_2)$ is equal to δ(state(c_1), inscr(head(c_1)), inscr(head(c_2)), state(c_2), head(c_2)$-$ head(c_1)).

Else $a(c_1, c_2) = 0$.

Using $\Delta = \{x \in \mathbb{C} : \exists (q_1, \sigma_1, \sigma_2, q_2, d) \in Q \times \Sigma \times \Sigma \times Q \times \{-1, 0, +1\} \wedge \delta(q_1, \sigma_1, \sigma_2, q_2, d) = x\}$ we can rewrite the ASM program to

forall $c : c \in C_M$ **do**
$S(c) := \sum_{\delta\in\Delta} \sum \{\!\{ S(c') \cdot \delta : c' \in C_M : \text{state}(c) = q \wedge \text{inscr}(\text{head}(c)) = \sigma \wedge$
$\text{state}(c) = q \wedge \text{inscr}(\text{head}(c)) = \sigma \wedge$
$\delta(q, \sigma, \sigma', q', \text{head}(c') - \text{head}(c)) = d \wedge$
$\forall i((i \neq \text{head}(c')) \rightarrow (\text{inscr}(c, i) = \text{inscr}(c', i)))\}\!\}$
endforall

Note that Δ is a finite set and determined by the Turing machine M. Therefore, the sum $\Sigma_{\delta\in\Delta} \ldots$ can be replaced by a sequence of binary sums.

Furthermore, the above program contains the formula $\forall i((i \neq \text{head}(c')) \rightarrow (\text{inscr}(c, i) = \text{inscr}(c', i)))$. This quantification might be problematic as it speaks about infinitely many elements. But we can avoid this by only considering the cells up to the length of the inscription (as only configurations are in the background which do not contain word of infinite length). Another possibility would be to add a successor relation $\text{Succ} \subseteq C_\text{M} \times \Sigma \times \{-1, 0, 1\} \times Q \times C_\text{M}$ to the background such that $\text{Succ}_1 \sigma d c_2$ holds if, and only if, starting from c_1, replacing the symbol at position of the head in c_1 by σ and moving the head in direction d leads to c_2.

Measurement. Until now, the simulation of a quantum Turing machine has been deterministic without measuring. The remaining question is, how to simulate a measurement in the given framework. Here, we describe one possibility for a simulation. In a state, a measurement is always performed with respect to an orthogonal partitioning $(C_i)_{i\in I}$, $I \subseteq \mathbb{N}$, of the underlying Hilbert-space. The probability to measure a component, can be directly calculated from the amplitudes of the elements in the component. The probability to observe the component C_i is $\sum_{x\in C_i} |S(x)|^2$. To simulate a measurement, we roughly proceed as follows.

1. Fix an order for the components (therefore $C_1, C_2, \ldots$ with probabilities $p_1, p_2, \ldots$. to observe)
2. Choose randomly a real number r from the interval $[0, 1)$
3. Return S_i if $r \in [\sum_{j=1}^{i-1} p_j, \sum_{j=1}^{i} p_j)$

As the interval indicating the choice of C_i has length p_i (one can also argue via the Lebesgue-measure) and as we suppose an equal distribution for the random choice of the real number, C_i is chosen with probability p_i.
The first step does not need to be realized in the ASM program. It can be integrated in the background of the states by a mapping $C : I \rightarrow \mathcal{P}$ where I is the set of indices for the components C_i. The reason is that the orthogonal partitioning belongs to the simulated QTM.

The nondeterministic ASM $R_{measure}^{(C(i))_{i\in I}}$ formalizes this process of measurement.

choose $x \in \mathbb{R} : 0 \leq x < 1$
  forall $i \in I$ **do**
    if $\sum_{j=1}^{i-1} \sum \{\!\{|S(x)|^2 : x \in C(j) : S(x) \neq 0\}\!\} \leq x <$
      $\sum_{j=1}^{i} \sum \{\!\{|S(x)|^2 : x \in C(j) : S(x) \neq 0\}\!\}$ **then**
        do-in-parallel
          forall $n \in C(i)$ **do**
            $S(n) := \frac{S(n)}{\sum \{\!\{|S(j)|^2 : j \in C(i) : S(j) \neq 0\}\!\}}$
          endforall
          forall $n \notin C(i)$ **do**
            $S(n) := 0$
          endforall
        enddo
      endif
    endforall
  endchoose

This ASM never produces an inconsistent update set as the intervals assigned to the components are disjoint for different components.

16.2.2 Quantum Circuits

We have introduced quantum circuits in chapter 15. Therefore, we give only the simulations by ASMs.

Simulating quantum circuits

As for quantum Turing machines, the description of a simulating ASM is partitioned into its basic components.

Static background and dynamic functions. The background of an ASM simulating a quantum circuit must include at least the following:

- $(\mathbb{C}, +, \cdot)$, the complex numbers with addition and multiplication
- multisets with union and intersection
- the set $\{0,1\}^*$ of finite words over $\{0,1\}$
- a function symbol : $\mathbb{N} \times \{0,1\}^* \to \{0,1\}$

The idea of this construction is that every word in $\{0,1\}^*$ represents a pure state of the underlying quantum state.

Furthermore, the basis of the circuits (the set of available gates) is included in the background. As this set might be infinite, we parameterize the mappings. Every basic gate with n inputs (and n outputs) corresponds to a matrix that is defined by a mapping $U : \{1,\ldots,2^n\} \times \{1,\ldots,2^n\} \to \mathbb{C}$. For coding the basis of the circuits, insert a mapping gate : $\mathbb{N}\times\mathbb{N}\times\mathbb{N} \to \mathbb{C}$ into the background with the following meaning. gate(k,i,j) is the value of the k-th basic gate for values i and j (if i and j exceed the number of lines respectively columns in the corresponding matrix then the value is undefined).

A dynamic function $S : \{0,1\}^* \to \mathbb{C}$ mainly characterizes the current state. The idea is that for $w \in \{0,1\}^*$, $S(w)$ is the amplitude of the pure state $|w\rangle$ in a superposition of pure states.

Simulating the work of a single quantum gate embedded into a circuit. Consider a quantum gate with m inputs (and m outputs) and the circuit with n inputs (and n outputs) containing this gate only once. Let $j_1 < \cdots < j_m$ be the indices of the qubits used by the gate. If the work of the gate on m qubits is specified by the $2^m \times 2^m$-matrix U then the work of the circuit on the n qubits is given by the $2^n \times 2^n$-matrix $\tilde{U}$ whose entries are defined as follows:

$$\begin{array}{lcll} \tilde{U}(|\overline{x}\rangle,|\overline{y}\rangle) & = & 0 & \text{if } \exists k \notin \{j_1,\ldots,j_m\}\ x_k \neq y_k \\ \tilde{U}(|\overline{x}\rangle,|\overline{y}\rangle) & = & U(|x_{j_1}\ldots x_{j_m}\rangle,|y_{j_1}\ldots y_{j_m}\rangle) & \text{else} \end{array}$$

The work of the above quantum circuit is simulated by the ASM program

forall $\overline{x} \in \{0,1\}^n$ **do**
$S(\overline{x}) := \sum\{\!\{ S(\overline{y}) \cdot U(x_{j_1}\ldots x_{j_m}, y_{j_1}\ldots y_{j_m}) : \overline{y} \in \{0,1\}^n : \forall k \notin \{j_1,\ldots,j_m\} y_{j_k} = x_{j_k} : \text{true} \}\!\}$
endforall

A possibly easier way to formulate the above is the following. Let $\Pi_{i,j}$ be the matrix corresponding to the permutation only exchanging the i-th and the j-th qubit. Furthermore, let U' be the $2^n \times 2^n$-matrix corresponding to the circuit that applies U to first m qubits.

$$\begin{array}{lcll} \tilde{U}(|\overline{x}\rangle,|\overline{y}\rangle) & = & 0 & \text{if } \exists k > m (x_k \neq y_k) \\ \tilde{U}(|\overline{x}\rangle,|\overline{y}\rangle) & = & U(|x_1\ldots x_m\rangle,|y_1\ldots y_m\rangle) & \text{else} \end{array}$$

Then $\tilde{U} = \Pi_{1,j_1}\Pi_{2,j_2}\ldots\Pi_{m,j_m}U'\Pi_{1,j_1}\Pi_{2,j_2}\ldots\Pi_{m,j_m}$.

The ASM program. An arbitrary quantum circuit is composed of a finite number of gates connected via wires. Accordingly, we compose an ASM simulating a circuit from the ASMs simulating the embedded gates.

In order to give an ASMs simulating the work of a complete quantum circuit, we consider the work of such a circuit closer. Its work can be divided into four types of steps:

1. the initialization
2. classical computation steps
3. quantum steps (namely transformations carried out by the gates)
4. measurement steps

The classical computation steps do not need to be considered closer as they can be simulated by an appropriate ASM according to the sequential and the parallel ASM thesis. This does also hold for the initialization.

The measurement can be simulated analogously to the one in quantum Turing machines. Therefore, the measurement for circuits would essentially be a repetition without new insights. Consequently, we leave out this part.

Only the quantum steps need to be considered a bit closer. The input of a quantum circuit is a quantum state, a superposition of pure states. All states with amplitude unequal to zero correspond to elements over $\{0,1\}^*$ of the same length. Therefore, a quantum circuit deals only with $\{0,1\}^n$ for some $n \in \mathbb{N}$. Using

$$\begin{aligned} \text{level}(1) &:= \text{the set of gates whose incoming wires are only input wires} \\ \text{level}(i) &:= \text{the set of gates whose incoming wires are only wires coming} \\ &\qquad \text{from gates of some gate from level}(j), j < i \end{aligned}$$

the work of a circuit with maximal level k is simulated by the ASM rule

forall $i \in \{1, \ldots, k\}$ **do**
  if level $= i$ **then**
    do-in-parallel
      ... all ASM rules for gates in level(i) ...
    enddo
  endif
endforall

The use of **forall** can be avoided by just writing down the k rules connected via **do-in-parallel** as every finite, acyclic circuit can be divided into a finite number of levels.

The ASM rules for gates can be simulated as already described where the indices of the incoming edges are parameters.

A further point is that if we go from the classical computation steps to quantum ones or vice versa then we have to convert the classical information into quantum information or extract some classical information from the quantum information. In the classical computation steps we only deal with pure states. I.e., we are only dealing with one word and e.g., go to its successor. If we change to quantum steps (with current word w) then just go to the state with $S(w) = 1$ and $S(w') = 0$ for all $w' \neq w$.
The other direction can only be handled via a measurement. To continue with classical steps, measure with respect to the orthogonal partitioning where each word corresponds to one component of the partitioning. Therefore, after such a measurement, the state collapses into a pure state respectively into a state where $S(w) = 1$ for exactly one $w \in \{0,1\}^*$ and $S(w') = 0$ for all $w' \neq w$.
For identifying the current kind of computation steps (classical, quantum, measurement, conversion), introduce some nullary function symbol Mode respectively having the value classical, measure, quantum.

16.2.3 Shor's Factorization Algorithm

In this section, we demonstrate how to simulate Shor's factorization algorithm by ASMs. We start with a short presentation of the algorithm. It proceeds within six steps.

The input of the algorithm is any natural number $n \in \mathbb{N}$. The algorithm can be shortly formulated as follows.

Algorithm:

1. If n is even then output 2 end

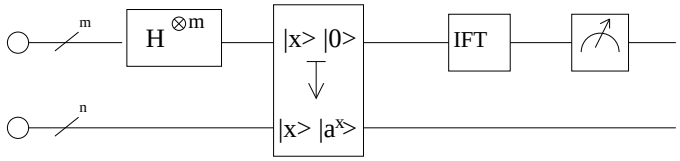
2. If $n = a^k$ for some $a \in \mathbb{N}$, $k \geq 2$ then output a end

3. Choose randomly $a < n$

   $d := \gcd(a, n)$

If $d > 1$ output d end

4. Let $m \in \mathbb{N}$ such that $n^2 \leq 2^m < 2n^2$ and
   r be the period of the function $k \mapsto a^k (\text{mod } k)$.
   Using the circuit



   we compute

$$|\varphi\rangle = \frac{1}{2^m} \sum_{l=0}^{r-1} \sum_{y=0}^{2^m-1} e^{\frac{2\pi i y l}{2^m}} \sum_{q=0}^{\max\{j \in \mathbb{N} : jr+1 \leq m\}} e^{\frac{2\pi i y r q}{2^m}} |y\rangle |a^l\rangle$$

   and measure the first m qubits to receive $y \in \mathbb{Z}_{2^m}$

5. Compute the convergents $\frac{p_i}{q_i}$ of $\frac{y}{2^m}$ and

   determine the smallest q_i such that $a^{q_i} \equiv 1 (\text{mod } n)$

   if such a q_i exists then $r := q_i$ else output ? end

6. If a^r is odd or $a^{\frac{r}{2}} \equiv -1 (\text{mod } n)$ then output ? end

   else compute $d := \gcd(n, a^{\frac{r}{2}} - 1)$ and output d end

The success probability of Shor's algorithm is at least $\Omega(\frac{1}{\log \log n})$.

The reason for the popularity of this algorithm is that its complexity is in $O((\log n)^3)$ but no classical algorithm is known that can factorize numbers in time $O((\log n)^k)$ for any k. Public key cryptosystem as RSA rely on the assumption that there is no classical algorithm breaking this bound and therefore, that it is not possible to crack them in polynomial time. By contrast, using Shor's algorithm we would be able to crack RSA in polynomial time.

Simulating Shor's factorization algorithm

As the structure of Shor's algorithm is a bit too complex to give a simulation at once and this simulation would not really give new insights. Instead of this, we give simulations for the basic components of Shor' s factorization algorithm. These are the Hadamard transformation, the quantum Fourier transformation

and Shor's algorithm for finding the order r of an element x in the group (mod n) (i.e., the least integer r such that $x^r = 1 \pmod{n}$). It has been proven that using randomization, factorization can be reduced to finding the order of an element. This is used in Shor's factorization algorithm and therefore, the main part of Shor's factorization algorithm is the order computation.

Hadamard transformation. For $x \in \{0,1\}^n$, the Hadamard transformation H_n of $|x\rangle$ is defined as follows.

$$H_n : |x\rangle \to \tfrac{1}{\sqrt{2^n}} \textstyle\sum_{y\in\{0,1\}^n} (-1)^{x\cdot y}|y\rangle$$

where $x \cdot y = \bigoplus_{i=1}^{n} x_i y_i$. We now describe an ASM executing the Hadamard transformation.

Background. Again the background contains $(\mathbb{C}, +, \cdot)$, a mapping σ assigning to every finite multiset of complex numbers the sum of its members multiplied with its multiplicity within the multiset respectively, $(\mathbb{R}, <)$, $(\mathbb{N}, <)$ and $\{0,1\}^*$.

Program. We start with rewriting the Hadamard-transformation such that it can be directly translated into an ASM program.
Consider a state (which is a superposition of pure states $|x\rangle$ for $x \in \{0,1\}^*$) with amplitude $S(x)$ for every pure state $|x\rangle$. In the next state (i.e., after having carried out the Hadamard-transformation), the amplitudes of the pure states are given by

$$\begin{array}{lcll} S_{new}(x) & = & \frac{1}{2^n} \sum_{y\in\{0,1\}^n} (-1)^{x\cdot y} \cdot S(y) & \text{if } x \in \{0,1\}^n \\ S_{new}(x) & = & S(x) & \text{else} \end{array}$$

The Hadamard transformation is computed by the following ASM rule $R^{n,m,l}_{Hadamard}$. More precisely, the following program reflects the case of a system with l registers where the Hadamard transformation is applied to the m-th register. For a pure state s, $\mathrm{Reg}_j(s) \in \{0,1\}^*$ is the content of the j-th register.

forall $k : k \in X_1 \times \cdots \times X_{m-1} \times \{0,1\}^* \times X_{m+1} \times \cdots \times X_l \wedge \mathrm{Reg}_j(k) \leq 2^n$ **do**
  forall $p \in \mathbb{R} : p \geq 0 \wedge p^2 = 2^n$ **do**
    $S(k) := \frac{1}{p} \sum_{k=0}^{q-1} \{\!\{ \frac{1}{2^n} \sum_{y\in\{0,1\}^*} (-1)^{\mathrm{Reg}_m(k)\cdot y} \cdot S(y) :: \bigwedge_{i\in\{1,\ldots,n\}, i\neq m} \mathrm{Reg}_m(k) \}\!\}$
  endforall
endforall

Quantum Fourier Transformation. The Quantum Fourier transform (QFT) with the base q (or in the group $\mathbb{Z}_q$) is the unitary transformation

$$\mathrm{QFT}_q : |a\rangle \to \tfrac{1}{\sqrt{q}} \textstyle\sum_{c=0}^{q-1} e^{2\pi i a c/q}|c\rangle$$

for $0 \leq a < q$. We now describe an ASM executing the quantum Fourier transformation.

Background. Again the background contains $(\mathbb{C}, +, \cdot)$, a mapping σ assigning to every finite multiset of complex numbers the sum of its members multiplied with its mulitplicity within the multiset respectively, $(\mathbb{R}, <)$ and $(\mathbb{N}, <)$.

Program. First, rewrite the Fourier-transformation such that it can be directly translated into an ASM program.
Consider a state (which is a superposition of pure states $|n\rangle$ for $n \in \mathbb{N}$) with amplitude $S(x)$ for every pure state. In the next state (i.e., after having carried out the Fourier transformation), the amplitudes of the pure states are given by

$$\begin{array}{lcll} S_{new}(|n\rangle) & = & S(|n\rangle) & \text{if } n \geq q \\ S_{new}(|n\rangle) & = & \frac{1}{\sqrt{q}} \sum_{k=0}^{q-1} e^{2\pi i k n/q} \cdot S(k) & \text{else} \end{array}$$

The QFT is computed by the following ASM rule $R_{QFT}(q)$ where q is a free variable. More precisely, the following program reflects the case of a system with l registers where the Fourier transformation is applied to the m-th register.
For a pure state s, $\mathrm{Reg}_j(s)$ is the content of the j-th register.

forall $k : k \in X_1 \times \cdots \times X_{m-1} \times \mathbb{N} \times X_{m+1} \times \cdots \times X_l \wedge \mathrm{Reg}_j(k) \leq q$ **do**
  forall $p \in \mathbb{R} : p \geq 0 \wedge p^2 = q$ **do**
    $S(k) := \frac{1}{p} \sum_{k=0}^{q-1} \{\!\{ e^{2\pi i \mathrm{Reg}_m(c) n/q} \cdot S(c) : c \in \mathbb{N} : 0 \leq Reg_m(c) \leq q-1 \wedge \bigwedge_{i \in \{1,\ldots,n\}, i \neq m} \mathrm{Reg}_m(k) \}\!\}$
  endforall
endforall

Order computation. In this subsection, we consider Shor's algorithm for finding the order r of an element x in the multiplicative group (mod n); that is, the least integer r such that $x^r = 1(\mod n)$.
The states resp. the vocabulary must subsume the requirements for the simulation of the Hadamard-transformation and the QFT. Let C be $\mathbb{N}^5$ representing the set of pure states (and $\mathbb{N}$ and $\{0,1\}^*$ are identified in the following).

Given an m-bit integer n, choose first a $q \in \mathbb{N}$ with $n^2 \leq q < 2n^2$ and q is a power of 2. Then start with five registers in states $|n, x, q, \overline{0}, \overline{0}\rangle$ where the last two registers have $\{\!\{\log n\}\!\}$ qubits.
The algorithm repeats the following steps $O(\log \log n)$ times:

1. Apply the **Hadamard transformation** to the fourth register.
   This leads to a state $\frac{1}{\sqrt{q}} \sum_{a=0}^{q-1} |n, x, q, a, \overline{0}\rangle$.
2. Using quantum parallelism, compute $x^a \mod n$ for all a in one step and store the result in the fifth register.
   This leads to a state $\frac{1}{\sqrt{q}} \sum_{a=0}^{q-1} |n, x, q, a, x^a \mod n\rangle$.

3. Apply the **quantum Fourier transformation** with parameter q to the fourth register.

4. **Measure** the fourth register.

Step 2 is simulated by the ASM rule $\tilde{R}$:

forall $z \in \mathbb{N}^5$ **do**
$S(z) := \sum\{\!\{S(z) : \bigwedge_{i=1}^{4} \mathrm{Reg}_i(y) = \mathrm{Reg}_i(z) \wedge\ \mathrm{Reg}_5(y) = \overline{0} \wedge$
$\mathrm{Reg}_5(z) = x^{\mathrm{Reg}_4(y)}(\mod n) : \mathrm{true}\}\!\}$
endforall

From the ASMs defined above, we can compose an ASM simulating the period computation:

do-in-parallel
if phase $= 0 \wedge$ counter $<$ upperBound **then**
choose $q \in \mathbb{N} : n^2 \leq q < 2n^2 \wedge \exists z \in \mathbb{N} \wedge q = 2^z$
forall $c \in \mathbb{N}^5$ **do**
if $\mathrm{Reg}_1(c) = n \wedge \mathrm{Reg}_2(c) = x \wedge \mathrm{Reg}_3(c) = q \wedge$
$\mathrm{Reg}_4(c) = \overline{0} \wedge \mathrm{Reg}_5(c) = \overline{0}$ **then**
$S(c) := 1$
else $S(c) := 0$
endif
endforall
endchoose
endif
if phase $= 1$ **then** $R_{Hadamard}^{\log_2 q,4,5}$ **endif**
if phase $= 2$ **then** $\tilde{R}$ **endif**
if phase $= 3$ **then** $R_{QFT}^{4}(q)$ **endif**
if phase $= 4$ **then** $R_{measure}^{(C_i)_{i\in\mathbb{N}}}$ **endif**
if phase < 4 **then** phase $:=$ phase $+ 1$ **endif**
enddo

where C_i consists of all 5-tuples in $\mathbb{N}$ where the fourth component is equal i. Furthermore, in an initial state phase $= 0$ has to be true.

16.2.4 Grover's Search Algorithm

In this section, we consider Grover's search algorithm treating the following problem:

> Given an unsorted database (list) of N items there is one satisfying a given condition, retrieve it.

More precisely, we use the following slight modification introduced in [18]. It is assumed that a function $f : \{0,1\}^* \to \{0,1\}$ is given as a black box such that $f(x_0) = 1$ for a single x_0.
Grover's search algorithm succeeds in the following steps:

1. Apply the **Hadamard transformation** H_n.
   This leads to a state $|\Phi\rangle = \frac{1}{\sqrt{2^n}} \sum_{a=0}^{2^n-1} |x\rangle$.
2. Apply the sign-changing operator V_f to $|\Phi\rangle$.
   This leads to the state $|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{a=0}^{2^n-1} (-1)^{f(x)} |x\rangle$.
3. Apply $D_n = -H_n R_n^1 H_n$.
4. Iterate $\{\!\{\frac{\pi}{4}\sqrt{2^n}\}\!\}$ times steps 2 and 3.
5. Measure the x register to get x_0. If $f(x_0) \neq 1$, go to step 1.

where R_n^1 is defined as follows:

$$\begin{array}{lcll} R_n^1[i,j] & = & 0 & \text{if } i \neq j \\ R_n^1[1,1] & = & -1 & \\ R_n^1[i,i] & = & 1 & \text{if } 1 < i \leq 2^n \end{array}$$

The states resp. the vocabulary must subsume the requirements for the simulation of the Hadamard-transformation and the QFT ($\mathbb{N}$ and $\{0,1\}^*$ are identified in the following).
Step 2 is simulated by the following ASM R_f:

forall $x \in \mathbb{N}$ **do**
  $S(x) := (-1)^{f(x)} \cdot S(x)$
endforall

Step 3 is simulated by the following ASM $\tilde{\tilde{R}}$:

do-in-parallel
  if subphase $= 1$ **then**
    forall $x \in \{0,1\}^n$ **do**
      $S(x) := (-1) \cdot S(x)$
    endforall
  endif
  if subphase $= 2$ **then** $R^n_{Hadamard}$ **endif**
  if subphase $= 3$ **then** $S(1) := (-1) \cdot S(1)$ **endif**

if subphase $= 4$ **then** $R^{n,1,1}_{Hadamard}$ **endif**
if subphase < 4 **then** subphase := subphase $+ 1$ **endif**
enddo

From the ASMs already defined, we can compose the following ASM simulating Grover's search algorithm:

do-in-parallel
if phase $= 1$ **then** $R^{n,1,1}_{Hadamard}$ **endif**
if phase $= 2$ **then** R_f **endif**
if phase $= 3$ **then** $\tilde{\tilde{R}}$ **endif**
if $\neg(\text{counter}^2 \geq \frac{\pi^2}{16}2^n) \wedge$ phase $= 4$ **then**
do-in-parallel
phase := 2
subphase := 1
counter := counter $+ 1$
enddo
endif
if (phase $\neq 4 \wedge$ phase $\neq 6$) **then** phase := phase $+ 1$ **endif**
if phase $= 5$ **then** $R^{(C_x)_{x\in\{0,1\}^*}}_{measure}$ **endif**
if phase $= 6$ **then**
if $\forall x(S(x) \neq 0 \rightarrow f(x) \neq 1$ **then**
do-in-parallel
phase := 1
subphase := 1
enddo
endif
endif
enddo

where $C_x = \{x\}$ for $x \in \{0,1\}^n$.
In an initial state counter must be equal to 0. phase and subphase must be equal to 1 respectively.

17 Postulates for Quantum Algorithms

Having given a description of one (albeit rather general) model of quantum algorithms by ASMs the question arises if also in the quantum case, the ASM thesis can be derived from some basic postulates that all models for quantum computation satisfy.

We describe such postulates here. As in the case of sequential and parallel algorithms, we need the Sequential Time Postulate (S1) and the Abstract State Postulate (S2). The Uniform Bounded Exploration Postulate (S3) will have to be modified and we will need postulates describing unitary transformation steps and measurement steps. Concerning the probabilism inherent in quantum algorithms we follow the convention that probabilistic algorithms have a function random in their vocabulary that takes as values real numbers in the interval $[0, 1]$. Update sets $\Delta(A, X)$ that do not depend on the value of random are called deterministic, otherwise they are probabilistic. The value of random is supposed to be changed uniformly, independently and externally (i.e. not by the algorithm) at any step. Another possibility would be to modify the sequential time postulate so that the next step transformation is of the form $\tau_A : \mathcal{S}(A) \times [0, 1] \to \mathcal{S}(A)$.

We also need a variant of the Background Postulate, tailored for quantum algorithms, which gives some minimal requirements on what the states should contain, and which also imposes a closure condition.

Recall that a state of a computation should contain everything that is necessary to determine the future progress of that computation. Hence a state of a quantum algorithm does not merely consist of an element of an appropriate Hilbert space, but also of a classical state containing information on the unitary operations used, and of course also on the classical parts of the computation. Hence we essentially adapt the notion of state from [5, 21] by augmenting it with an element $|\psi\rangle$ of a Hilbert space. Sticking to the format of first-order structures, such an element is formally described by a function $\Psi : B \to \mathbb{C}$ from the basis B of the Hilbert space to the complex numbers. We use the notation $X = (X_c, |\psi\rangle)$ for the states of a quantum algorithm where X_c is the *classical part* of the state X (everything except Ψ).

Q1: Quantum State Postulate. Besides the conventions from [21], all states contain the three constants classical, quantum, and measure (which are interpreted by distinct elements) and a nullary function mode whose value

is one of these three elements. Further all states contain a nullary function symbol basis whose value is a finite or countable set B, and a function symbol Ψ whose value is a function from B to the complex numbers $\mathbb{C}$, describing a unit length element $|\psi\rangle$ of the Hilbert space with orthonormal basis B. Further the state space $\mathcal{S}(A)$ of any quantum algorithm is closed under variations of $|\psi\rangle$, i.e., if $(X_c, |\psi\rangle) \in \mathcal{S}(A)$ then also $(X_c, |\varphi\rangle) \in \mathcal{S}(A)$ for every unit length $|\varphi\rangle$ of the same Hilbert space.

The function mode divides the state space into three classes, the classical states, the quantum states, and the measure states, each satisfying different postulates on the next step transformation.

The remaining postulates should ensure the following properties: For quantum states $(X_c, |\psi\rangle)$, the update of $|\psi\rangle$ is defined by a unitary transformation U (that depends only on X_c and A but not on $|\psi\rangle$). For measurement states $(X_c, |\psi\rangle)$ the update of $|\psi\rangle$ is defined by a (probabilistic) projection. Finally, the classical part X_c of any state $(X_c, |\psi\rangle)$ is updated by a sequential algorithm. The only unbounded parallelism of a quantum algorithm is the simultaneous update of all amplitudes in unitary transformation and measurement steps.

Remark. One might criticize this last requirement as too restricted: why not consider "quantum parallel algorithms"? The answer is that the very point of quantum algorithms is to use quantum effects in order to implement with polynomial resources what classically requires exponential parallelism. Of course one could define computation models that combine quantum computing with unbounded classical parallelism, but this seems not interesting since we lose the advantage of quantum algorithms without going beyond the power of classical parallel algorithms.

It is relatively straightforward to formulate postulates for the first two properties.

Q2: Unitary Transformation Postulate. If $X = (X_c, |\psi\rangle)$ is a quantum state then $\tau_A(X)$ is either a quantum or a measure state, with the same value of basis. The change of $|\psi\rangle$ from X to $\tau_A(X)$ is described by a unitary operator U_X (which only depends on X_c).

Q3: Measurement Postulate: If $X = (X_c, |\psi\rangle)$ is a measure state then the change of $|\psi\rangle$ from X to $\tau_A(X)$ is described by a collection $\{M_{X,j} : j \in J\}$ of linear operators (which only depends on X_c) such that $\sum_{j\in J} M^*_{X,j} M_{X,j} = I$ and the index set J is linearly ordered.

The value of $|\psi\rangle$ after the measurement step is described by

$$\frac{M_{X,j}|\psi\rangle}{\sqrt{\langle\psi|M^*_{X,j}M_{X,j}|\psi\rangle}}$$

for one j, the probability for each j being $p(j) = \langle\psi|M^*_{X,j}M_{X,j}|\psi\rangle$.

Remark. When we say that a linear operator W defining the update of $|\psi\rangle$ in a state $X = (X_c, |\psi\rangle)$ depends only on X_c, we mean that the same operator W applies to every $X' = (X_c, |\psi'\rangle)$ with the same classical part X_c.

It is a little more delicate to formulate in a precise, yet natural way that the only parallelism is inside unitary transformation and measurement steps. In fact there are several possibilities.

We split the update sets of a quantum algorithm A on state $X = (X_c, |\psi\rangle)$ into the classical updates of the functions in X_c and the updates of $|\psi\rangle$, i.e., $\Delta(A, X) = \Delta(A, X_c) \cup \Delta(A, |\psi\rangle)$. A high-level formulation of the sequentiality postulate might be the following.

Q4: Sequentiality Postulate. For every quantum algorithm A, the classical update sets and all entries of the unitary transformation matrices and the projection matrices at quantum states and measure states are computed by sequential algorithms.

Here is a more explicit version of postulate Q4.

Q4': Sequentiality Postulate (a detailed version). For every quantum algorithm A, there exist sequential algorithms A_1, A_2, A_3 with the following properties:
(1) A_1 has the same states as A and at every state $X = (X_c, |\psi\rangle)$ of A, the classical update set $\Delta(A, X_c)$ equals $\Delta(A_1, X)$, and in the case that X is a classical state, $\Delta(A_1, X)$ is the entire update set $\Delta(A, X)$,
(2) A_2 has states (X_c, x, y) consisting of the classical part of a quantum state X of A, base vectors x, y in basis, and the set of outputs $\mathrm{out}_{A_1}((X_c, x, y))$ contains exactly the entry $U_X(x, y)$ of the unitary transformation U_X.
(3) A_3 has states (X_c, x, y, j) consisting of the classical part of a measure state X of A, base vectors x, y in basis, and $j \in J$, and the set of outputs $\mathrm{out}_{A_1}((X_c, x, y, j))$ contains exactly the entry $M_{X,j}(x, y)$ of the projection matrix $M_{X,j}$ at state X.

The same postulate can also be formulated in a different way, similar to the Uniformly Bounded Exploration Postulate for sequential algorithms.

Q4'': Limited Exploration Postulate. Let $\Upsilon = \Upsilon_c \cup \{\Psi\}$ be the vocabulary of the quantum algorithm A. There exist finite sets T_1, T_2, T_3 of terms with the following properties.
(1) T_1 is a set of ground terms over vocabulary Υ and whenever two states

X and X' coincide on T_1, then $\Delta(A, X_c) = \Delta(A, X'_c)$, and if X and X' happen to be classical states, then even $\Delta(A, X) = \Delta(A, X')$.
(2) T_2 has vocabulary Υ_c and for all $t \in T_2$, free$(t) \subseteq \{x, y\}$. Whenever X and X' are two quantum states of A, a, b are elements of basis in X, and a', b' are elements of basis in X', such that (X_c, a, b) and (X'_c, a', b') coincide on T_2, then the entries $U_X(a, b)$ and $U_{X'}(a', b')$ of the appropriate unitary transformations are equal.
(3) T_3 has vocabulary Υ_c and for all $t \in T_2$, free$(t) \subseteq \{x, y, j\}$. Whenever X and X' are two measure states of A, a, b are elements of basis in X, and a', b' are elements of basis in X', such that (X_c, a, b, k) and (X'_c, a', b', k') coincide on T_3, then the entries $M_{X,k}(a, b)$ and $M_{X',k'}(a', b')$ of the appropriate projections are equal.

Example. In section 16.2.1, we introduced quantum Turing machines as an example of a quantum computation model and gave a simulation by means of ASM. Now, we explain exemplarily for QTMs why they fit into the above postulates. The sequential algorithm computing for every state X the entries $U_X(c, c')$ of the associated unitary transformation returns $\delta(q_1, \sigma_1, \sigma_2, q_2, d)$ in the case that σ_1 is the content of c' at the position of the head, q_1 is the state of c' and c results from c' by replacing in c' the symbol at the position of the head with σ_2, the state of c' with q_2 and moving the head according to d. Otherwise, it returns 0. Note that for each c there are only finitely many non-zero values $U(c, c')$. Concerning measurement steps in a quantum Turing machine, the set J in the postulate Q3 is the set of configurations C_M. The sequential algorithm demanded in postulate Q4', returns for a basis vector b the value 1 if it is in the subspace spanned by the basis vector v and 0 otherwise.

18 The Equivalence Theorem

In order to simplify the proof of the equivalence theorem, we give a normal for sequential ASM in advance.

Lemma 18.1. For every sequential ASM rule there exists an equivalent ASM rule of the form

do-in-parallel
  if φ_1 **then** R_1 **endif**
  if φ_2 **then** R_2 **endif**
  $\vdots$
  if φ_n **then** R_n **endif**
enddo

where R_i is an update rule of the form $s_i := t_i$ or an output-rule Output(t_i).

Proof. We prove the lemma by effectively constructing for an arbitrary sequential ASM an equivalent sequential ASM of the described form. The main point in the proof are the following two equivalence preserving rewriting rules.

original rule	**equivalent rule**

if φ **then**
  do-in-parallel
  Π_1
  Π_2
  enddo
endif
	do-in-parallel
	  if φ **then** Π_1 **endif**
	  if φ **then** Π_2 **endif**
	enddo

if φ **then**
  if ψ **then** $\tilde{\Pi}$ **endif**
endif
	if $\varphi \wedge \psi$ **then** $\tilde{\Pi}$ **endif**

By successively applying these rules until no more rule fits to the current ASM, we obtain an equivalent sequential ASM of the claimed form. □

Theorem 18.2. Every quantum algorithm is equivalent to an ASM working with the same background.

Proof. As already mentioned the states of a quantum algorithm can be partitioned into three classes: classical states, quantum states and measurement states.

Classical states. Postulate Q4' and the sequential ASM thesis imply that there exists an ASM A_c whose update set equals the classical update set of the quantum algorithm on any of its states. In the case of a classical state, the update set of A_c is the entire update set.

Quantum states. Postulate Q4' and the sequential ASM thesis for ASMs with output imply that there exists an ASM whose output for every quantum state X and every pair of base vectors $b, c \in$ basis contains exactly the entry $U_X(b, c)$. Further, there is an equivalent ASM with a program of the form given in Lemma 18.1 and with $\text{free}(\varphi_i), \text{free}(t_i) \subseteq \{b, c\}$. Since we are only interested in the output we can assume that all rules R_i, $i \in \{1, \ldots, n\}$ are output-rules of the form $\text{Output}(t_i)$. The following ASM A_q simulates the execution of the unitary transformation induced by the quantum algorithm in the case of a quantum state (w.l.o.g., we assume that $\bigvee_{i=1}^{n} \varphi_i$ is valid and for all $k, l \in \{1, \ldots, n\}, k \neq l$ the formula $\varphi_k \wedge \varphi_l$ is not satisfiable):

forall $b : b \in$ basis **do**
  $\Psi(b) := \sum_{i=1}^{n} \sum \{\!\{ t_i(b, c) \cdot \Psi(c) : c \in \text{basis} : \varphi_i(b, c) \}\!\}$
enddo

Measure States. With analogous arguments, we infer that there exists an ASM with a program of the form given in Lemma 18.1 whose output for every measure state X, every pair of base vectors $b, c \in B$ and every $j \in J$ contains exactly the entry $M_{X,j}(b, c)$. The rules R_i, $i \in \{1, \ldots, n\}$, are output-rules of the form $\text{Output}(t_i)$ and $\text{free}(\varphi_i), \text{free}(t_i) \subseteq \{b, c, j\}$. The following ASM A_m simulates the measurement induced by the quantum algorithm in the case of a measure state and an already chosen value for random.

forall $b, i : b \in \text{basis} \wedge i \in J$ **do**
  if $\ell(i) \leq \text{random} < \ell(i) + p(i)$ **then**
    $\Psi(b) := \frac{1}{\sqrt{n(i)}} \sum_{j=1}^{n} \sum \{\!\{ t_j(b, c) \cdot \Psi(c) : c \in \text{basis} : \varphi_j(b, c, i) \}\!\}$
  endif
enddo

where

$\ell(j)$ is the probability that the subspace belonging to a k with $k < j$ is measured (the set J is linearly ordered by postulate Q3).

$p(j)$ is the probability that the subspace belonging to j is measured.

$n(j)$ is the inverse of the square of the renormalization factor. It is obtained by summing the $\Psi_{\text{new}}(b)^* \cdot \Psi_{\text{new}}(b)$ for all base-vectors b where Ψ_{new} is the amplitude function after executing the transformation M_j and before carrying out normalization.

Hence $\ell(j), p(j), n(j)$ can be defined as follows:

$$\begin{aligned} \ell(j) &= \sum \{\!\{ p(i) :: i \in J \wedge v < j \}\!\} \\ p(j) &= \sum \{\!\{ \Psi(b)^* \cdot \Psi_{\text{new}}(b) : b \in \text{basis} : \text{true} \}\!\} \\ n(j) &= \sum \{\!\{ \Psi_{\text{new}}(b)^* \cdot \Psi_{\text{new}}(b) : b \in \text{basis} : \text{true} \}\!\} \end{aligned}$$

with $\Psi_{\text{new}}(b) = \sum_{j=1}^{n} \sum \{\!\{ t_j(b,c) \cdot \Psi(c) : b \in \text{basis} : \varphi_j(b,c,j) \}\!\}$. The desired ASM combines A_c, A_q, and A_m as follows:

do-in-parallel
  A_c
  if mode = quantum **then** A_q **endif**
  if mode = measure **then** A_m **endif**
enddo

This completes the proof of the theorem. □

Remark. From the proof of Theorem 18.2, we see that it is not necessary to allow arbitrary parallelism in the ASMs that simulate quantum algorithms. The classical update sets are generated by a sequential ASM. Unbounded parallelism does appear in the simulation of the quantum steps and the measure steps, but in a very restricted way. We have only one forall-instruction (without nestings) and an update rule involving a uniformly bounded number of summations over a multiset. Comprehension terms are not nested (the ASM given in the proof uses nested comprehension terms but it can be rewritten without nestings). Hence, the equivalence theorem could be formulated in a stronger way by allowing only ASMs satisfying the above constraints.

There are also many cases where multisets are not needed. One example are quantum Turing machines. In this case, every base vector (i.e. every configuration) is connected via a non-zero entry of the transformation matrix to a uniformly bounded number of other base vectors (depending only on the algorithm, not on the input). Summations over the multisets for the quantum states can thus be replaced by sums over a uniformly bounded number of elements.

Bibliography

[1] Abstract State Machines. `http://www.eecs.umich.edu/gasm/`. Website of the University of Michigan, maintained by J. Huggins.

[2] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational Transducers for Electronic Commerce. In *Proceedings of 17th ACM Symposium on Principles of Database Systems (PODS '98)*, pages 179–187. ACM Press, 1998.

[3] H. Andréka, J. van Benthem, and I. Németi. Modal Languages and Bounded Fragments of Predicate Logic. *Journal of Philosophical Logic*, 27:217–274, 1998.

[4] D. Beauquier and A. Slissenko. A First Order Logic for Specification of Timed Algorithms: Basic Properties and a Decidable Class. *Annals of Pure and Applied Logic*, 113(1-3):13–52, 2002.

[5] A. Blass and Y. Gurevich. Abstract State Machines Capture Parallel Algorithms. *ACM Transactions on Computational Logic*, 4(4):578–651, 2003.

[6] E. Börger, P. Päppinghaus, and J. Schmid. Report on a Practical Application of ASMs in Software Design. In *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 361–366. Springer-Verlag, 2000.

[7] E. Börger and R. Stärk. *Abstract State Machines*. Springer-Verlag, 2003.

[8] I. Chuang and M. Nielsen. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[9] A. Dold. A Formal Representation of Abstract State Machines Using PVS. Technical Report 6.2, Universität Ulm, July 1998. Verifix Project.

[10] A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In *International Workshop on Abstract State Machines ASM 2000*, volume 1912 of *LNCS*, pages 303–332. Springer-Verlag, 2000.

[11] A. Gargantini and E. Riccobene. ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science*, 7(11), 2001.

[12] G. Gottlob, E. Grädel, and H. Veith. Datalog LITE: A deductive query language with linear time model checking. *ACM Transactions on Computational Logic*, 3(1):1–35, 2002.

[13] E. Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 64:1719–1742, 1999.

[14] E. Grädel. The decidability of guarded fixed point logic. In J. Gerbrandy, M. Marx, M. de Rijke, and Y. Venema, editors, *JFAK. Essays Dedicated to Johan van Benthem on the Occasion of his 50th Birthday*. Amsterdam University Press, 1999.

[15] E. Grädel. Guarded fixed point logic and the monadic theory of trees. *Theoretical Computer Science*, 288:129–152, 2002.

[16] E. Grädel and A. Nowack. Quantum Computing and Abstract State Machines. In *Abstract State Machines - Advances in Theory and Applications*, volume 2589 of *LNCS*, pages 309 – 323. Springer-Verlag, 2003.

[17] E. Grädel and I. Walukiewicz. Guarded Fixed Point Logic. In *Proceedings of 14th IEEE Symposium on Logic in Computer Science LICS '99, Trento*, pages 45–54, 1999.

[18] J. Gruska. *Quantum Computing*. McGraw-Hill, 1999.

[19] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[20] Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report Technical Report CSE-TR-336-97, EECS Dept, University of Michigan, 1997.

[21] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1:77–111, 2000.

[22] Y. Gurevich. Logician in the land of OS: Abstract State Machines in Microsoft. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, pages 129–136, 2001.

[23] M. Hirvensalo. *Quantum Computing*. Springer, 2001.

[24] I. Hodkinson. Monodic Packed Fragment with Equality is Decidable. *Studia Logica*, 72(2):185–197, September 2002.

[25] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 106:85–134, 2000.

[26] R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. PhD thesis, TU München, 1998.

[27] A. Nowack. Deciding the Verification Problem for Abstract State Machines. In *Abstract State Machines - Advances in Theory and Applications*, volume 2589 of *LNCS*. Springer-Verlag, 2003.

[28] A. Nowack. Slicing abstract state machines. In *Abstract State Machines 2004. Advances in Theory and Practice*, volume 3052 of *LNCS*, pages 186–201. Springer-Verlag, 2004.

[29] P. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26:1484–1509, 1997.

[30] M. Spielmann. Automatic Verification of Abstract State Machines. In *Proceedings of 11th International Conference on Computer-Aided Verification (CAV '99)*, volume 1633 of *LNCS*, pages 431–442. Springer-Verlag, 1999.

[31] M. Spielmann. Model Checking Abstract State Machines and Beyond. In *International Workshop on Abstract State Machines ASM 2000*, LNCS, pages 323–340. Springer-Verlag, 2000.

[32] M. Spielmann. Verification of Relational Transducers for Electronic Commerce. In *19th ACM Symposium on Principles of Database Systems PODS 2000, Dallas*, pages 92–93. ACM Press, 2000.

[33] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[34] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[35] J. van Benthem. Dynamic Bits and Pieces. Technical Report LP-97-01, ILLC, University of Amsterdam, 1997.

[36] L. Vandersypen, M. Steffen, G. Breyta, C. Yannoni, M. Sherwood, and I. Chuang. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414, 2001.

[37] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

[38] F. Wolter and M. Zakharyaschev. Decidable Fragments of First-Order Modal Logics. *Journal of Symbolic Logic*, 66(3):1415–1438, 2001.

[39] F. Wolter and M. Zakharyaschev. Axiomatizing the monodic fragment of first-order temporal logic. *Annals of Pure and Applied Logic*, 118(3):133 – 145, 2002.

Index

Curriculum Vitae

Zur Person

Name:	Antje Erna Helga Nowack
Geboren:	12. 3. 1976 in Düsseldorf

Bildungsgang

1982 - 1986	Grundschule (Gottfried-Kricker-Grundschule in Willich-Anrath)
1986 - 6/1995	Erasmus-von-Rotterdam-Gymnasium in Viersen
6/1995	allgemeine Hochschulreife
10/1995 - 7/2000	Diplom-Studiengang an der RWTH Aachen
14. 7. 2000	Diplom in Informatik
9/2000 - 9/2004	Wissenschaftliche Angestellte am Lehr- und Forschungsgebiet *Mathematische Grundlagen der Informatik* (Professor Dr. Erich Grädel), RWTH Aachen