A Guarded Fragment for Abstract State Machines

Antje Nowack

Mathematische Grundlagen der Informatik,
RWTH Aachen, D-52056 Aachen, Germany,
email: nowack@informatik.rwth-aachen.de

Abstract. Abstract State Machines (ASMs) provide a formal method for transparent design and specification of complex dynamic systems. They combine advantages of informal and formal methods.
Applications motivate a number of computability and decidability problems connected to ASMs. Such problems result for example from the area of verifying properties of ASMs. Their high expressive power leads rather directly to undecidability respectively uncomputability results for most interesting problems in the case of unrestricted ASMs. Consequently, it is rather natural to ask whether there exist expressive classes of ASMs for which we can prove positive decidability and computability results.
In this work, we introduce such a class of ASMs. The idea of this class is the same as for the guarded fragment of first-order logic. We analyse the expressive power of this class and prove that it is stronger than Datalog LITE and the guarded fragment of first-order fixed point logic. Some decidability and computability results have been proven in earlier works.

1 Introduction

Abstract State Machines (ASMs) have been introduced as an attempt 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 complex 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. There is a precise semantics and this allows the application of mathematical methods to analyse ASMs. In particular, these are often results from mathematical logic as ASMs use classical mathematical structures to describe states of a computation.

Anyhow, ASMs are easily understandable. ASM specifications are easy to read and to write. And this is one of the most important (pre)conditions for a broad acceptance in practice and a resulting application. The reason for the understandability is that ASMs have a very simple syntax that is quite similar to pseudo-code. And indeed, ASMs are used in practice to specify hardware and software systems. Reports on this can be found e.g. in [3].

Furthermore, a number of computability and decidability problems is connected to ASMs mostly motivated by applications. Examples for decidability problems are different variants of the verification problem. The general verification problem - for example - deals with the question whether all computations of an ASM satisfy a property given in some formalism (usually, some temporal logic). An example for computability problems is the computation of a minimal slice. A slice of an ASM consists of a subset of the rules of the original ASM. On a specified part of the states, this slice behaves in exactly the same way as the original ASM. The motivation for computing a minimal slice of an ASM comes from debugging. If an error is observed, the programmer tries to extract that part of the ASM which is responsible for the error. This part might be much smaller than the original ASM.

On the one hand, ASMs are a very powerful model of computation and this is one of their important advantages. In order to witness this, it has been proven that well-defined classes of ASMs simulate sequential, parallel and quantum algorithms on their natural abstraction level (see [11], [2], [8]).

On the other hand, the high expressive power leads rather directly to undecidability resp. uncomputability results for most interesting problems in the case of unrestricted ASMs. For instance, the general verification problem is not decidable and a minimal slice is not computable.

Consequently, it is rather natural to ask whether there exist expressive classes of ASMs for which we can prove positive decidability and computability results. In this work, we introduce such a class of ASMs. Some decidability and computability results have already been proven in other works (see [12], [13]). We analyse the expressive power of this class and prove that it is stronger than Datalog LITE and the guarded fragment of first-order fixed point logic.

Besides the above aspects, we expect our results to increase the understanding of Datalog LITE and the guarded fragment of first-order fixed point logic.

Outline. First, we give a short introduction to ASMs. After briefly considering some decision and computation problems, we introduce the class of guarded (deterministic) ASMs denoted by GF($\mathcal{D}$). In the remainder of this work, we prove that GF($\mathcal{D}$) is stronger than Datolog LITE and the guarded fragment of first-order fixed point logic.

2 A Brief Background on Abstract State Machines

In this work, $\mathcal{D}$ denotes the class of deterministic ASMs. Readers that are common with ASMs please note that we do not allow the use import (and do not consider it in the remainder). As we restrict our considerations to these ASMs, we give only a short introduction to the class $\mathcal{D}$ and do not present further constructs. For detailed information on ASMs (formerly called evolving algebras), see e.g. [10].

The basic idea of ASMs is the following. The states of an ASM are finite first-order structures. An ASM is defined via a program that determines the transitions between the states. The basic instructions 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** $\bar{x}$: 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 the guard φ evaluates to true. The inner rule of a forall-rule is executed in parallel for all tuples of elements satisfying the guard φ of the forall-rule. 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. This yields a sequence of structures called a run. At first, a run is infinite as an ASM program can always be applied to an appropriate structure. But we can cut off a run starting from a certain point in case that there are no more changes in the states. We say that the ASM halts.

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.

Formally, the syntax of ASM rules defining an ASM from $\mathcal{D}$ is given in figure 1. The notion of term is defined as usual. As we consider relations as special functions (i.e., functions that can only develop to true or to false), the outermost symbol of a term can also be a relation symbol. Terms whose outermost symbol is a relation symbol are called boolean-valued. Consequently, such terms can only evaluate to true or to false.

An ASM rule R defines for every state $\mathcal{A}$ and every variable assignment ξ a set of updates resp. an update set $\mathrm{US}(R, \mathcal{A}, \xi)$ (see figure 2).

State. A state of an ASM is a finite first-order structure over the same vocabulary as the program defining the ASM.

Update. An update of a state $\mathcal{A}$ is a tuple $\beta = (f, \bar{a}, b)$ where f is a function symbol of arity r, $\bar{a}$ is an r-tuple of elements from the domain of $\mathcal{A}$ and b is an element from the domain of $\mathcal{A}$. Executing this update means to change the interpretation of f such that the value of f at $\bar{a}$ becomes b and the interpretation of f for the other tuples is left as before. Note that if f is a relation symbol then b must be contained in the set $\{\text{true}, \text{false}\}$.

Executing in parallel all elements of $\mathrm{US}(\Pi, \mathcal{A}, \xi)$ leads to the successor state of $\mathcal{A}$ with respect to Π. A problem arises if two updates in an update set contradict each other.

Let Υ be a finite vocabulary (i.e., a finite set of relation and function symbols). The class of ASM rules over Υ is defined by induction as follows.

Skip Skip is an ASM rule.

Update rule For all terms s 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 be boolean-valued.

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

Block If R_1 and R_2 are rules then **do-in-parallel** R_1 R_2 **enddo** is a rule.

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

Free and bounded variables are defined as usual with **forall** $\bar{x} : \varphi$ binding $\bar{x}$. A program is a rule without free variables.

Fig. 1. Syntax of ASM rules over the finite vocabulary Υ

Contradictory. A deterministic update set U is contradictory if it contains a pair of updates $(f, \bar{a}, b)$ and $(f, \bar{a}, c)$ where $b \neq c$; otherwise U is consistent.

To execute a consistent update set $\{(f_1, \bar{a}_1, b_1), \ldots, (f_k, \bar{a}_k, b_k)\}$, set $f\bar{a}_1, \ldots, f\bar{a}_k$ to $b_1, \ldots, b_k$ respectively and leave the rest of the interpretations as they are.

To execute a contradictory deterministic update set, do nothing; the new state is identical to the old one.

Executing Π once for a state $\mathcal{A}$ leads to the successor of $\mathcal{A}$.

Successor. For a deterministic program Π, $\mathcal{A} \vdash_{\Pi} \mathcal{A}'$ holds if $\mathrm{US}(\Pi, \mathcal{A}, \xi)$ transforms the state $\mathcal{A}$ into the state $\mathcal{A}'$. The state $\mathcal{A}'$ is a successor of $\mathcal{A}$ with respect to Π.

The iterated execution of an ASM Π leads to a run of Π.

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. Then 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 $A_0 = I$.

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 halts (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 $\mathcal{A}$.

The evaluation of a deterministic rule R at a state $\mathcal{A}$ under a variable assignment ξ produces a deterministic update set $U = \mathrm{US}(R, \mathcal{A}, \xi)$. To execute R, execute the update set that is defined by induction on the rule:

- $\mathrm{US}(\mathrm{Skip}, \mathcal{A}, \xi) = \emptyset$
- $\mathrm{US}(f(t_1, \ldots, t_r) := t_0, \mathcal{A}, \xi) = \{((f, [\![t_1]\!]_\xi^\mathcal{A}, \ldots, [\![t_r]\!]_\xi^\mathcal{A}), [\![t_0]\!]_\xi^\mathcal{A})\}$
- $\mathrm{US}(\textbf{if } g \textbf{ then } R' \textbf{ endif}, \mathcal{A}, \xi) = \begin{cases} \mathrm{US}(R', \mathcal{A}, \xi) & \text{if } \mathcal{A}, \xi \models g \\ \emptyset & \text{else} \end{cases}$
- $\mathrm{US}(\textbf{do-in-parallel } R_1 \ R_2 \textbf{ enddo}, \mathcal{A}, \xi) = \mathrm{US}(R_1, \mathcal{A}, \xi) \cup \mathrm{US}(R_2, \mathcal{A}, \xi)$
- $\mathrm{US}(\textbf{forall } \bar{z} : \varphi \ R' \textbf{ endforall}, \mathcal{A}, \xi) = \bigcup_{\bar{a} \in M} \mathrm{US}(R', \mathcal{A}, \xi[\bar{z} \to \bar{a}])$

For a term t, $[\![t_1]\!]_\xi^\mathcal{A}$ denotes the value of t in $\mathcal{A}$ with ξ.

Fig. 2. Semantics of ASM rules

3 Decidability and Computability Problems

As already mentioned in the introduction, there is a number of decidability and computability problems connected to ASMs. In this section, we consider two examples. They are both motivated by the application of ASMs in software engineering as a specification formalism. As writing formal specifications is usually not an error free process, we are rather early concerned with the task of finding errors in a specification. 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. The second approach is to test ASMs. The executabilty yields this possibility. We do not consider testing itself but a technique that might be combined with testing (but can also be combined with verification). This technique is called slicing.

The verification problem. The aim of verifying ASMs is to prove that the behaviour of an ASM specification coincides with the desired one. There are mainly two possible meanings for the question whether "the behaviour of an ASM specification coincides with the desired one":

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

While the restricted verification problem is decidable for $\mathcal{D}$ and formulae from temporal first-order logic, the general verification problem is undecidable in this case.

The reason for the decidability of the restricted verification problem is the following. As the domain of the states does not change in a run of an ASM from

$\mathcal{D}$, every run of an ASM in $\mathcal{D}$ contains only a finite number of different states. An upper bound for this number can be immediately computed from the initial state. It suffices to consider the intial part of the run up to a length coreresponding to this upper bound in order to descide whether a run satisfies a property expressed in linear temporal first-order logic. So, the restricted verification problem is comparable to a model-checking problem with finite space.

The reason for the undecidability of the general verifcation is that the (finite) satisfiability problem for first-order logic can be immediately reduced to the general verification problem. Nevertheless, it is possible to find decidable instances. One of them consists of a well-defined fragment of the class GF($\mathcal{D}$) of ASMs (see [12]).

Slicing. In order to debug or to modify an ASM specification, we are most often not interested in its complete behaviour but only in a specified part of it. Consider for instance the case of debugging. When an error is observed, the developer tries to extract that part of the specification which is responsible for the erroneous behaviour. This set of rules 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.

This observation motivates the concept of slicing. By applying this technique, one extracts rules from an ASM program that are relevant for the behaviour with respect to the specified parts of the states. The result is a slice which is a syntactically correct ASM again. The smaller such a slice is the better.

Consider for example an ASM involving a relation R. If we are only interested in the evaluation of R then we can cut off all rules from the ASM that do not influence it.

Although a minimal slice is not computable for arbitrary ASMs from $\mathcal{D}$, it is computable for ASMs from the fragment GF($\mathcal{D}$) of $\mathcal{D}$ (see [13]) which we consider in the following.

4 The Guarded Fragment

Both problems presented in the preceding section are strongly connected to satisfiability problems in mathematical logic. In order to demonstrate this connection, we consider the ASM defined by the program $\Pi :=$ **if** φ **then** res := true **endif** where φ is an arbitrary first-order formula not involving the nullary relation symbol res.

Every computation of Π satisfies the temporal formula (X res) (stating that res holds in the next state) iff $\neg\varphi$ is not finitely satisfiable.

Similarly, computing a minimal slice for an ASM from $\mathcal{D}$ is directly connected to the satisfiability problem for first-order logic. **Skip** is a minimal slice of Π relative to the evaluation of res iff 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 induces the undecidability of the general verification problem and non-computability of a minimal slice for ASMs from $\mathcal{D}$.

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 consider decidable instances of the (finite) satisfiability problem and to try to obtain decidable resp. computable instances of decision resp. computation problems for ASMs by introducing similar restrictions for ASMs. In the past, a number of concepts has been presented in order to obtain decidable instances of the (finite) satisfiability problem. Examples are the so-called prefix classes, restrictions on the vocabularies or the two-variable fragment. In most cases, a transfer of these concepts is not suitable as the corresponding restrictions on the ASMs are very strong.

In [1], an approach to decidable instances of the satisfiability problem for first-order logic has been presented that is essentially different from the ones mentioned above. It originates from an investigation of the reason for the good model-theoretic properties of propositional modal logic (ML). H. Andréka, J. van Benthem and I. Németi have considered the translation from ML to first-order logic a bit closer and detected that in the modal fragment, all quantifications are guarded by some atomic formula. This observation then lead to the definition of the guarded fragment of first-order logic.

Definition 1 *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 $\psi(\bar{x}, \bar{y})$ is a formula in GF then $\exists \bar{y}(\alpha(\bar{x}, \bar{y}) \wedge \psi(\bar{x}, \bar{y}))$ and $\forall \bar{y}(\alpha(\bar{x}, \bar{y}) \rightarrow \psi(\bar{x}, \bar{y}))$ belong to GF provided that $\mathrm{free}(\psi) \subseteq \mathrm{free}(\alpha) = \bar{x} \cup \bar{y}$ and $\alpha(\bar{x}, \bar{y})$ is a relational atomic formula.*

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

Based on a number of results (see [1], [5]), 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. However, most of the interesting properties are still expressible in GF. The following definition transfers the idea of GF to the ASMs from $\mathcal{D}$.

Definition 2 *The guarded fragment* $\mathrm{GF}(\mathcal{D})$ *of the class $\mathcal{D}$ of ASMs is the subset of $\mathcal{D}$ such that every element Π of* $\mathrm{GF}(\mathcal{D})$ *uses only relation and constant symbols and satisfies the following conditions:*

- *the lefthand side and the righthand 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** $\bar{x} : \alpha(\bar{x}, \bar{y})$ R **endforall** *where* $\alpha(\bar{x}, \bar{y})$ *is an atomic formula and* $\mathrm{free}(R) \subseteq \mathrm{free}(\alpha) = \{\bar{x}, \bar{y}\}$

Note that the definition of the guarded fragment of $\mathcal{D}$ can be generalized to any class $\mathcal{C}$ of ASMs (also nondeterministic).

5 Basic properties of Guarded ASMs

In section 3, we have already addressed some properties of $\mathrm{GF}(\mathcal{D})$ concerning decidability respectively computability. In this section, we consider some basic properties with regard to the expressive power. The aim of this section is to increase the understanding of $\mathrm{GF}(\mathcal{D})$.

To formulate some basic properties of guarded ASMs, we give the definition of the Gaifman graph of a structure in advance. The Gaifman graph of a structure $\mathcal{A}$ is an undirected graph with the same domain as $\mathcal{A}$. There is an edge between two elements of the domain iff they coexist in some atomic fact in $\mathcal{A}$.

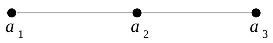
Definition 3 *The Gaifman graph of a relational structure $\mathcal{A}$ (with universe A) is the undirected graph $G(\mathcal{A}) = (A, E^{\mathcal{A}})$ where*

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

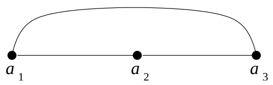
An immediate consequence of the definition of $\mathrm{GF}(\mathcal{D})$ is that we can not access pairs of elements that are not connected via an edge in the Gaifman graph. This fact implies the following lemma.

Lemma 4 *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.*

As a consequence of this lemma, we obtain some non-computability results. E.g., the transitive closure is not computable by an ASM from $\mathrm{GF}(\mathcal{D})$. In order to prove this, consider the class of undirected graphs. In this case, the Gaifman graph of a state $\mathcal{A}$ is obtained 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 $(a_1, a_3) \in E^{\mathcal{B}}$).

To disprove the converse of lemma 4 we use the notion of equivalence of ASM in the following sense.

Definition 5 *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. $\mathcal{A} \vdash_{\Pi_1} \mathcal{B}$ if, and only if, $\mathcal{A} \vdash_{\Pi_2} \mathcal{B}$.*

Lemma 6 *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). Consider the ASM defined by

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

As we have only negative updates in Π, 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 $E^{\mathcal{B}} \subseteq E^{\mathcal{A}}$.

Now, we prove that there is no ASM $\Pi' \in$ GF($\mathcal{D}$) that is equivalent to Π. The proof is done by contradiction. Assume that there exists an ASM $\Pi' \in$ 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 **if** ψ **then** $P :=$ false **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 lefthand 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 the parallel composition of **if** t **then** $P :=$ true **endif** and **if** $\neg t$ **then** $P :=$ false **endif**. 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 **if** $\neg t$ **then** $P :=$ false **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 equivalence preserving rewriting rules given in figure 3 to Π' leads to an ASM from GF($\mathcal{D}$) that is equivalent to Π' and thus to Π. 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.

original rule	equivalent rule

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

Fig. 3. Rewriting rules applied in the proof of lemma 6

Though we have seen some limits of GF($\mathcal{D}$) in this section, many interesting properties are nevertheless definable in GF($\mathcal{D}$). We demonstrate that in the remainder of this work.

6 Guarded ASMs are Stronger than Datalog LITE

In this section, we compare GF($\mathcal{D}$) to Datalog LITE (see [4]). 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 [4], 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 the restriction to the alternation-free fragment is not needed for comparing GF($\mathcal{D}$) to guarded fixed-point logic. This restriction has been necessary for the comparison of Datalog LITE to guarded fixed-point logic. Furthermore, we even obtain a strict result in the case of GF($\mathcal{D}$).. I.e., GF($\mathcal{D}$) is stronger than guarded (first-order) fixed point logic. Though this proposition already implies that GF($\mathcal{D}$) is stronger than Datalog LITE, we prove it elementarilly. We expect to obtain a better insight to the connection between Datalog LITE and GF($\mathcal{D}$) in this way as the proof is much more straightforward.

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 predicate 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 $(\Pi, Q)[\mathcal{A}]$, ie., 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 valuated, we can compute the interpretation of the head prediates 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 $\neg S v_1 \ldots v_k$ 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 $\forall y_1 \ldots y_n (\alpha \rightarrow \beta)$ where α and β are atoms, and $\mathrm{free}(\beta) \subseteq \mathrm{free}(\alpha)$. The free variables $\mathrm{free}(G)$ of G are given by $\mathrm{free}(\alpha) - \{y_1, \ldots, y_n\}$.

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 $\mathrm{GF}(\mathcal{D})$ is strictly more expressive than Datalog LITE. The notion of ASM query allows to compare ASMs and Datalog LITE directly. An ASM query (Π_{ASM}, Q) is equivalent to a Datalog LITE query $(\Pi_{\mathrm{Datalog}}, Q)$ if, and only if, for every state $\mathcal{A}$ (over an appropriate vocabulary) and every tuple $\bar{a}$ of elements from the domain of $\mathcal{A}$

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

Consequently, we consider equivalence with respect to the result of a computation. Another possibility would be to compare programs stepwisely as we did it for two ASMs. This means that 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 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 7 *Every Datalog LITE query is equivalent to a* GF($\mathcal{D}$) *query. Conversely, there exists a* GF($\mathcal{D}$) *query for which there is no equivalent Datalog LITE query.*

Proof. First, we show that 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 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 these 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.

The first step in the translation is to map every Datalog LITE rule R to an ASM rule $a(R) \in$ GF($\mathcal{D}$). 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 in B with free variable 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 $\mathrm{free}(g) = \{\bar{x}\}$, α be its head and B its body. Then $a(R)$ is the rule

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

where $\bar{x}$ 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 frmom 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** $a(R_1) \ldots a(R_n)$ **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 $\mathrm{computed}_0, \ldots, \mathrm{computed}_{r-1}$ marking whether the computation of the i-th stratum is completed. Initially, $\mathrm{computed}_0, \ldots, \mathrm{computed}_{r-1}$ are all interpreted by false. D is translated to the ASM defined by

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

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

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

and the formulae $\varphi_i, i \in \{0, \ldots, r\}$, 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

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

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

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

In the rest of this proof, we show that $\mathrm{GF}(\mathcal{D})$ is *strictly* stronger than Datalog LITE. I.e., there exists a $\mathrm{GF}(\mathcal{D})$ query for which there is no equivalent Datalog LITE query. In order to prove strictness, we use the following proposition that has been proved in [4].

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 $\mathcal{A}$. 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. A witness is the ASM $\Pi \in \mathrm{GF}(\mathcal{D})$ defined by **forall** $xy : Exy$ **do** $Exy :=$ false **endforall**. □

7 Guarded ASMs are Stronger than Guarded Fixed Point Logic

The most important extension of propositional modal logic (ML) is the modal 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 the guarded fixed point logic μGF.

μGF is the second mechanism to which we compare $\mathrm{GF}(\mathcal{D})$. It has first been considered in [9]. Further considerations can be found in [6] and [7].

Definition 8 *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 $\bar{x} = x_1 \ldots x_k$ be a tuples of distinct variables. Further, let $\psi(W, \bar{x})$ be a guarded formula where W appears only positively and not in guards. Moreover, we require that all the free variables of $\psi(W, \bar{x})$ are contained in $\bar{x}$. For such a formula $\psi(W, \bar{x})$, we can construct the formulae

$$\mathrm{lfp}_{\bar{x},W}\psi(\bar{x}) \textit{ and } \mathrm{gfp}_{\bar{x},W}\psi(\bar{x})$$

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 $\psi(W, \bar{x})$ defines an operator on k-ary relations $W \subseteq A^k$, namely

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

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

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

and

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

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

In the remainder of this section, we compare the expressive power of $\mathrm{GF}(\mathcal{D})$ to the expressive power of the guarded fixed point logic.

In [4], it has been proven that 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. This proposition and lemma 7 imply the following corollary.

Corollary 9 *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 sharpened. 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.

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

Definition 10 *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* $\bar{a}$ *be a tuple of elements from the domain of* $\mathcal{A}$ *and* R *be a relation symbol whose arity corresponds to the length of* $\bar{a}$*. Then we say that* $\bar{a} \in (\Pi, R)[\mathcal{A}]$ *holds iff* $\mathcal{A}^*_\Pi \models R\bar{a}$*.*

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 GF($\mathcal{D}$) and μGF and because of the inductivity 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 $\bar{a}$ of elements from the domain of $\mathcal{A}$

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

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

Theorem 11 *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 an equivalent GF($\mathcal{D}$)-query.

W.l.o.g., we can assume that different quantifactions 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, $\mathrm{free}(\varphi) \subseteq \mathrm{free}(\alpha_\varphi)$. In the case where φ is a sentence, we can set $\alpha_\varphi := \mathrm{true}$.

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 intitializing these relations with the empty set (resp. by false if they are nullary). Then our construction ensures that

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

for every structure resp. state $\mathcal{A}$ and every tuple $\bar{a}$ 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 $\mathcal{A}$ iff $\mathcal{A} \models \varphi(\bar{a})$.

Furthermore, we associate with every subformula φ of ψ, that is non-atomic or uses a fixed point variable, a relation $\mathrm{computed}_\varphi$. This relation has the same arity as φ and indicates for a tuple $\bar{a}$ 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 $\bar{a}$). For atomic formulae φ not involving a fixed-point variable, $\mathrm{computed}_\varphi \bar{x}$ 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:

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

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

Now, we define for every μGF-sentence φ an ASM $\Pi_\varphi \in$ GF($\mathcal{D}$) that is equivalent to φ in the already defined sense. Π_φ is defined by induction on φ. We partially omit **do-in-parallel** if it is understood, e.g. inside a forall-rule.

If φ is an atomic formula then Π_φ consists of the single rule

> **forall** $\bar{x} : \varphi(\bar{x})$ **do**
> $H_\varphi \bar{x} := \mathrm{true}$
> $\mathrm{computed}_\varphi \bar{x} := \mathrm{true}$
> **endforall**

If φ does not use a fixed-point variable then H_φ is initialized by the empty relation. Consequently, H_φ contains exactly those tuples of elements satisfying φ after executing the above updates.

Else, the valuation of φ may change. But we know that all fixed point operators are inductive. Therefore, it is not possible that a tuple is first contained 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 following rule where $\bar{x}'$ and $\bar{x}''$ are subtuples of $\bar{x}$.

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

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 $H_{\nu \wedge \vartheta}$ is the conjunction of the relations H_ν and H_ϑ.

If $\varphi = \neg\vartheta$ then Π_φ is the following rule where $\bar{x}'$ is a subtuple of $\bar{x}$.

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

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

If $\varphi = \forall \bar{y}(\alpha_\vartheta \rightarrow \vartheta)$ then Π_φ is the following rule where $\bar{x}'$ is a subtuple of $\bar{x}$, $\bar{x}''$ is a subtuple of $\bar{x}'$ and $\bar{z}'$ is a subtuple of $\bar{z}$

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

```
        computed_φ x̄' := true
      endif
    endforall
  enddo
```

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.

If $\varphi = [\mathrm{lfp}_{W,\bar{x}}\vartheta(W,\bar{x})](\bar{x})$ then, by the induction hypothesis, there is an ASM from GF($\mathcal{D}$) computing the update operator defined by $\vartheta(W,\bar{x})$. 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. That is, 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 x̄: α_φ(x̄, ȳ) do
    if computed_ϑ x̄ then     // computation with current values completed
      if ¬(W x̄ ↔ ϑ(x̄)) then      // fixed point not yet reached
        do-in-parallel
          if ϑ then W x̄ := true endif     // update of fixed point variable

           // Compute the valuation of those formulae that 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 variable constructor outermost then the respective
           // fixed point has to be computed new. Until this computation is
           // not finished (i.e., the fixed point is not yet reached) the value
           // of the respectively computed is false for all important tuples.

          forall ū_ξ1 : H_ξ1(ū_ξ1) do H_ξ1(ū_ξ1) := false endforall
           // ξ_i has to be computed again with new interpretation of W
          forall z̄_ξ1 : α_ξ1(x̄^ξ1, z̄_ξ1) do computed_ξ1(x̄^ξ'1, 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^ξ'm, z̄'_ξm) := false endforall
          forall z̄_ρ1 : α_ρ1(x̄^ρ1, z̄_ρ1) do computed_ρ1(x̄^ρ'1, z̄'_ρ1) := false endforall
          ⋮
          forall z̄_ρn : α_ρn(x̄^ρn, z̄_ρn) do computed_ρn(x̄^ρ'n, z̄'_ρn) := false endforall
        enddo
      endif
    endif
  endforall
  if ∀x̄(α_φ(x̄, ȳ) → (W(x̄) ↔ ϑ(x̄))) then
```

forall $\bar{x} : \alpha_\varphi(\bar{x}, y)$ **do** $\text{computed}_\varphi(\bar{x}) := \text{true}$ **endforall**
endif
enddo

where $\bar{x}^{\xi_i}$ and $\bar{x}^{\rho_j}$ $(i \in \{1, \ldots, m\},\ j \in \{1, \ldots, n\})$ are subtuples of $\bar{x}$, $\bar{x}^{\xi_i'}$ is a subtuple of $\bar{x}^{\xi_i}$ $(i \in \{1, \ldots, m\})$, and $\bar{x}^{\rho_j'}$ is a subtuple of $\bar{x}^{\rho_j}$ $(j \in \{1, \ldots, n\})$. Furthermore $\bar{z}'_{\xi_i}$ is a subtuple of $\bar{z}_{\xi_i}$ $(i \in \{1, \ldots, m\})$ and $\bar{z}'_{\rho_j}$ is a subtuple of $\bar{z}_{\rho_j}$ $(j \in \{1, \ldots, n\})$.
Π_φ results from Π'_φ by replacing all occurences of W and H_ϑ by H_φ. 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 as described above, 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, $\text{computed}_\varphi \bar{x}$ has to be replaced by true.
Initial states are those which satisfy the following three formulae.

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

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

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

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 $\alpha := \text{false}$ and updates of the from $\alpha := \text{true}$ both in ASMs from GF($\mathcal{D}$). Therefore, the evaluation of the dynamic relations is not always inductive. It even does not need to halt. 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 one defined by the program

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

forall $\bar{x} : R\bar{x}$ **do**
  if $Q\bar{x}$ **then** $Q\bar{x} :=$ false **endif**
endforall
enddo

Second, the guards in μGF are not allowed to contain fixed-point variables. There is an extension of μGF (we denote it by $\mu\mathrm{GF}^e$) that admits to use fixed-point variables in the guards. M. Otto has proven that there is a $\mu\mathrm{GF}^e$-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 $\mu\mathrm{GF}^e$ is equivalent to a GF($\mathcal{D}$)-query. This implies immediately the above proposition. □

References

1. H. Andréka, J. van Benthem, and I. Németi. Modal Languages and Bounded Fragments of Predicate Logic. Technical Report ILLC Research Report ML-96-03, ILLC, 1996.
2. A. Blass and Y. Gurevich. Abstract state Machines Capture Parallel Algorithms. *ACM Transactions on Computation Logic*, 4(4):578–651, October 2003.
3. E. Börger, P. Päppinghaus, and J. Schmid. Report on a Practical Application of ASMs in Software Design. In Y. Gurevich, M. Odersky, and L. Thiele, editors, *International Workshop on Abstract State Machines ASM 2000*, volume 1912 of *LNCS*, pages 361–366. Springer-Verlag, 2000.
4. 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.
5. E. Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 64:1719–1742, 1999.
6. 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*, CD-ROM. Amsterdam University Press, 1999.
7. E. Grädel. Guarded fixed point logic and the monadic theory of trees. *Theoretical Computer Science*, 2001. To appear.
8. 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.
9. 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.
10. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
11. Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1:77–111, 2000.
12. 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.
13. A. Nowack. Slicing abstract state machines. In *Abstract State Machines 2004. Advances in Theory and Practice*, volume 3052 of *LNCS*. Springer-Verlag, 2004.