Slicing Abstract State Machines

Antje Nowack

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

Abstract. In this work, we introduce slicing for Abstract State Machines (ASMs). The idea of this concept is analogous to the one of program slicing which is an established technique for extracting statements from a program that are relevant for its behaviour 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 identify an expressive class of ASMs for which a minimal (static) slice is computable (and prove the computability). 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.

1 Introduction

In order to debug or to modify a program, we are most often not interested in its complete behaviour but only in its behaviour at a given point of interest. This means that we do not observe each complete program state but only a 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 behaviour. 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 behaviour 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 [6]. 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 [5] for an overview.

Abstract State Machines (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 [2]. 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 work can be summarized 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). In addition, it turns out that this class has a high expressive power. It captures a number of expressive formalisms well-known from literature.

Extensions. Once we have proven the basic computability result it is easy to obtain several extensions of it. We present some of them. First, we extend the class and explain how the basic result extends to these superclasses.

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

2 Definition of Slicing

As this paper is written for the ASM community, we assume that the reader is familiar with ASMs. For a detailed introduction to ASMs see e.g. [4].

If no restriction is given then the guard of an if-clause, forall-rule or choice-rule may be any first-order formula (possibly containing quantifiers).

In this section, 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 behaviour of a program considered in [6] is defined by the values of its program variables. For ASMs, a state is defined by the content of its locations (see [4]). Locations can be described by atomic formulae.

Definition 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 1 and figure 2 where $\mathrm{sub}(\psi)$ abbreviates the set of subformulae of a FO-formula ψ and $\mathrm{sub}(\Pi)$ 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 formulae) to the corresponding locations.

Definition 2 *Let $\mathcal{A}$ be a state and S be a slicing criterion.*

$L(\mathcal{A}, S)$ denotes the set of all locations $l = (Q, a_1 \dots a_r)$ in $\mathcal{A}$ such that the atom $Qa_1 \dots a_r$ is an instance of an atomic formula $\alpha \in S$.

ψ	**sub**(ψ)
atomic	$\{\psi, \text{true}, \text{false}\}$
$\neg\varphi$	$\{\psi\} \cup \text{sub}(\varphi) \cup \{\neg\vartheta : \vartheta \in \text{sub}(\varphi)\}$
$\varphi_1 \vee \varphi_2$	$\{\psi\} \cup \text{sub}(\varphi_1) \cup \text{sub}(\varphi_2) \cup \{\vartheta_1 \vee \vartheta_2 : \vartheta_1 \in \text{sub}(\varphi_1), \vartheta_2 \in \text{sub}(\varphi_2)\}$
$\varphi_1 \wedge \varphi_2$	$\{\psi\} \cup \text{sub}(\varphi_1) \cup \text{sub}(\varphi_2) \cup \{\vartheta_1 \wedge \vartheta_2 : \vartheta_1 \in \text{sub}(\varphi_1), \vartheta_2 \in \text{sub}(\varphi_2)$
$\exists\bar{x}\ \varphi$	$\{\psi\} \cup \text{sub}(\varphi) \cup \{\exists\bar{x}\ \vartheta : \vartheta \in \text{sub}(\varphi)\}$
$\forall\bar{x}\ \varphi$	$\{\psi\} \cup \text{sub}(\varphi) \cup \{\forall\bar{x}\ \vartheta : \vartheta \in \text{sub}(\varphi)\}$

Fig. 1. Formal definition of sub(ψ) for an FO-formula ψ

Π	**sub(Π)**
Skip	{**Skip**}
$s := t$	{**Skip**, Π}
if φ **then** Π' **endif**	{**Skip**, Π} $\cup$ sub(Π') $\cup$ {**if** ϑ **then** $\tilde{\Pi}$ **endif** : $\tilde{\Pi} \in \text{sub}(\Pi'), \vartheta \in \text{sub}(\varphi)$}
do − in − parallel Π_1 Π_2 **enddo**	{**Skip**, Π} $\cup$ sub(Π_1) $\cup$ sub(Π_2) $\cup$ {**do − in − parallel** Π'_1 Π'_2 **enddo** : $\Pi'_1 \in \text{sub}(\Pi_1), \Pi'_2 \in \text{sub}(\Pi_2)$}
import v R' **endimport**	{**Skip**, Π} $\cup$ sub(R') $\cup$ {**import** v $\tilde{R}$ **endimport** : $\tilde{R} \in \text{sub}(R')$}
forall $\bar{x} : \varphi(\bar{x})$ **do** R' **endforall**	{**Skip**, Π} $\cup$ sub(R') $\cup$ {**forall** $\bar{x} : \vartheta(\bar{x})$ $\tilde{R}$ **endforall** : $\tilde{R} \in \text{sub}(R'), \vartheta \in \text{sub}(\varphi)$}
choose $\bar{x} : \varphi(\bar{x})$ **do** R' **endchoose**	{**Skip**, Π} $\cup$ sub(R') $\cup$ {**choose** $\bar{x} : \vartheta(\bar{x})$ $\tilde{R}$ **endchoose** : $\tilde{R} \in \text{sub}(R'), \vartheta \in \text{sub}(\varphi)$}

Fig. 2. Formal definition of sub(Π) for an ASM Π

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 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 $(R, \bar{a})$ in exactly the same way as Π where the atom $R\bar{a}$ is an instance of an atomic formula $\alpha \in S$.

Definition 4 (Slice) *Let Π be an ASM-program and S be a slicing criterion. An S-slice of Π is an ASM-program $\Pi_S \in \mathrm{sub}(\Pi)$ such that $\Pi \equiv_S \Pi_S$.*

Remark. The set of locations, on which 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 rest of this work, we refer to the size of a formula as the number of symbols in the string representing it where variables, constant and relation symbols 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), 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.

Note that in the following considerations, we could use any notion of size that is computable from an ASM or a formula.

Definition 5 (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 $|\tilde{\Pi}| < |\Pi_S|$.*

3 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 $\mathrm{sub}(\Pi)$ is finite for any ASM Π and Π itself is an S-slice for any slicing criterion S. Note that the finiteness of $\mathrm{sub}(\Pi)$ does not imply that a minimal slice is computable. In general, it is not decidable whether an element of $\mathrm{sub}(\Pi)$ 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 $\{\mathrm{Mode}\}$-slice of $\Pi =$ **if** φ **then** Mode := true **endif** iff φ is not satisfiable. Furthermore, **Skip** is the only minimal $\{\mathrm{Mode}\}$-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**

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).

4 Guardedness Condition for ASMs

For obtaining positive computability results, we have to put restrictions on the ASMs. In order to benefit from the computability, the expressive power of the classes resulting from these restrictions should be rather high.

One class satisfying these conditions is obtained by taking only those ASMs that satisfy some guardedness condition. Informally, every first-order quantifier (possibly appearing in some guard) and every appearance of **forall** or **choose** is relativized by an atomic formula. We have mentioned the choice of an element because the following definition is not only applicable to deterministic ASMs but to any class of ASMs. We have chosen this general definition as we will later also consider nondeterministic ASMs.

In the main part of this work, we consider deterministic ASMs not using **import**.

Definition 6 *$\mathcal{D}$ is the class of all deterministic ASMs not using* **import**.

The definition of the guarded fragment $\mathrm{GF}(\mathcal{C})$ of a class $\mathcal{C}$ of ASMs uses the guarded fragment of first-order logic, that has originally been introduced in [1].

Definition 7 (Guarded Fragment of First-Order Logic) *The guarded fragment (GF) of first-order logic is defined by induction as follows*

- *every relational atomic formula belongs to GF.*
- *GF is closed under propositional connectives* $\neg$, $\wedge$, $\vee$.
- *if* $\bar{x}$, $\bar{y}$ *are tuples of variables,* $\alpha(\bar{x}, \bar{y})$ *is an atomic formula and* φ *is a formula in GF such that* $\mathrm{free}(\varphi) \subseteq \mathrm{free}(\alpha) = \{\bar{x}, \bar{y}\}$, *then also the formulae* $\exists \bar{y}(\alpha(\bar{x}, \bar{y}) \wedge \varphi(\bar{x}, \bar{y}))$ *and* $\forall \bar{y}(\alpha(\bar{x}, \bar{y}) \rightarrow \varphi(\bar{x}, \bar{y}))$ *belong to GF.*

In GF, all quantifications are guarded by atomic formulae. Note that formulae from GF contain only relation and constant symbols. An important property of GF is that in spite of its high expressive power, the (finite) satisfiability problem is decidable for GF (see [1]).

Definition 8 *The guarded fragment* $\mathrm{GF}(\mathcal{C})$ *of a class* $\mathcal{C}$ *of ASMs is the subset of* $\mathcal{C}$ *such that every element* Π *of* $\mathrm{GF}(\mathcal{C})$ *uses only relation and constant symbols and satisfies the following conditions:*

- *the left-hand side and the right-hand side of any update rule is a boolean-valued term*
- *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* $\mathrm{free}(R) \subseteq \mathrm{free}(\alpha) = \{\bar{x}, \bar{y}\}$
- *every choice-rule (choice of an element from the domain) has the form* **choose** $\bar{x} : \alpha(\bar{x}, \bar{y})$ R **endchoose** *where* $\alpha(\bar{x}, \bar{y})$ *is an atomic formula and* $\mathrm{free}(R) \subseteq \mathrm{free}(\alpha) = \{\bar{x}, \bar{y}\}$

In the next two sections, we prove the main proposition of this work.

Theorem 9 *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*.*

In the rest of *this* section, we consider the expressive power of $\mathrm{GF}(\mathcal{D})$.

$\mathrm{GF}(\mathcal{D})$ has a rather high expressive power. In order to demonstrate this, we compare the expressive power of $\mathrm{GF}(\mathcal{D})$ to the expressive power of other well-known and expressive mechanisms.

The first such mechanism is Datalog LITE (see [3]). This is a guarded version of stratified Datalog additionally allowing a limited form of universal quantification in rule bodies.

Proposition 10 $\mathrm{GF}(\mathcal{D})$ *is strictly more expressive than Datalog LITE.*

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.

Furthermore, it has been proven in [3] that Datalog LITE is equivalent in expressive power to alternation-free guarded fixed point logic, a natural fragment of the guarded fixed point logic obtained by disallowing alternations between least and greatest fixed points. Guarded fixed point logic is the extension of GF by least and greatest fixed points. 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 capturing result.

Proposition 11 $\mathrm{GF}(\mathcal{D})$ *is strictly more expressive than guarded (first-order) fixed point logic.*

The intuitive reason why we can abandon the restriction of alternation-freedom is the following. 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 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 we can abandon alternation-freedom.

The above facts show us the expressive power of GF($\mathcal{D}$) but nevertheless, we might still miss an intuition about this class. In order to get a basic understanding of it, 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.

As a consequence of the above observation, we get some non-computability results. E.g., the transitive closure is not computable by an ASM from GF($\mathcal{D}$).

The above considerations demonstrate the high expressive power of GF($\mathcal{D}$). We defer a detailed investigation of the computational power of GF($\mathcal{D}$) to another paper.

5 Example

In this section, we exemplarily consider the ASM $\Pi \in$ GF($\mathcal{D}$) (see figure 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 iff 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, lifeness is equal to false and safe is equal to true. When the ASM halts (i.e., no more changes in the interpetations 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 iff all reachable states are safe and lifeness is true iff 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

do-in-parallel
  forall $x : Sx$ **do**
    $Rx :=$ true
  endforall
  forall $x : Rx$ **do**
    forall $ay : Taxy$ **do**
      if $\neg Ry$ **then** $Ry :=$ true **endif**
    endforall
  endforall
  forall $x : Rx$ **do**
    do-in-parallel
      if $\neg\exists ay\ Taxy$ **then** lifeness := false **endif**
      if $\neg Px$ **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

Fig. 3. ASM Π (left) and its minimal $\{Rx\}$-slice Π_R (right)

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 3). Though R influences the evaluation of safe and lifeness, 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. However, it might be intuitive to embed the update rule in a conditional 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.

do-in-parallel
  $\tilde{\Pi}_R$
  forall $x : Rx$ **do**
    if $\neg Px$ **then** safe := false **endif**
  endforall
enddo

where $\tilde{\Pi}_R$ results from Π_R by removing **do-in-parallel** and **enddo**.

From this slice, we see that a slice may involve updates of locations whose lefthand 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 lifeness are computed in parallel, the evaluation of

safe does not involve lifeness. Consequently, we can cut off all updates of lifeness in order to obtain a $\{\text{safe}\}$-slice of Π.
Note that the minimal $\{\text{safe}\}$-slice is equal to the minimal $\{\text{safe}, Rx\}$-slice.

6 Quasistates and Partial Equivalence

The main goal of this paper 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 behaviour on all runs resp. S-equivalence is reduced to the same behaviour on the runs of the standard states of the quasistates. In the proof of the computability result, the formulae occuring in the ASM are evaluated on quasistates. The idea of the notion of quasistate is similar to the one of quasimodel introduced in [1].

6.1 Quasistates

For a finite set F of first-order formulae (FO-formulae), an F-type is a maximal consistent subset of F.

For a sequence of variables $\bar{y}$, two types Δ and Δ' are called $\bar{y}$-close ($\Delta =_{\bar{y}} \Delta'$) if Δ and Δ' have the same formulae with free variables disjoint from $\bar{y}$.

Definition 12 (Quasistate) *Let F be a finite set of FO-formulae.*

An F-quasistate is a set of F-types T such that for each $\Delta \in T$ and each formula $\exists \bar{y} \varphi \in \Delta$ there is a type $\Delta' \in T$ with $\varphi \in \Delta'$ and $\Delta =_{\bar{y}} \Delta'$.

Φ holds in a quasistate if $\Phi \in \Delta$ for some Δ in this quasistate.

In particular, the $\bar{y}$-closeness implies for every sentence $\varphi \in F$ that $\varphi \in \Delta$ for all $\Delta \in Q$ or $\varphi \notin \Delta$ for all $\Delta \in Q$.

For an F-quasistate Q, we define a standard state $\mathcal{A}_Q$. The idea of this construction is that $\mathcal{A}_Q$ realizes exactly the quasistate 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 $\exists \bar{y} \varphi \in \Delta_i$, $\varphi \in \Delta_{i+1}$ and $\Delta_{i+1} =_{\bar{y}} \Delta_i$. We say that the variables in $\bar{y}$ 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 $\langle (\pi_j, x_j) \rangle_{j \in J}$ iff 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 subsection, we have explained how to 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 $Q^F_{\mathcal{A}}$ that 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 13 *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:*

$$Q^F_{\mathcal{A}} := \{\Delta : \Delta \text{ is an } F\text{−type that is realized in } \mathcal{A}\}$$

6.2 S-equivalence on Quasistates

In this subsection, we prepare the proof of our main result. First, we define for an ASM Π the set T_Π of all formulae whose valuation possibly influences the run of an ASM Π.

Definition 14 *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, 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 of those formulae in $\mathcal{A}$ that 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_1 and Π_2. All other formulae do not have any influence.
In the rest of this work, $Q^{\Pi_1,\Pi_2}_{\mathcal{A}}$ abbreviates $Q^{T_{\Pi_1,\Pi_2}}_{\mathcal{A}}$ and $Q^{\Pi}_{\mathcal{A}}$ abbreviates $Q^{T_\Pi}_{\mathcal{A}}$.

In the previous subsection, we have defined the F-quasistate of a state $\mathcal{A}$. Analogously, one can define the F-type of a tuple in a state $\mathcal{A}$.
Let F a finite set of FO-formulae, $\mathcal{A}$ be a state and $V = \{x_1, \ldots, x_n\}$ be the set of variables of F. Any tuple $\bar{a} = a_1 \ldots a_n$ of elements from the domain A of $\mathcal{A}$ induces a variable assignment $\sigma_{\bar{a}} : V \to A$ such that $\sigma_{\bar{a}}(x_i) = a_i$ for $i \in \{1, \ldots, n\}$ (the variable assignment induced by $\bar{a}$).

The T_{Π_1,Π_2}-type $\Delta^{\Pi_1,\Pi_2}_{\mathcal{A},\bar{a}}$ of $\bar{a}$ in $\mathcal{A}$ is the T_{Π_1,Π_2}-type $\Delta \in Q^{\Pi_1,\Pi_2}_{\mathcal{A}}$ that contains exactly those formulae $\varphi \in T_{\Pi_1,\Pi_2}$ with $\mathcal{A}, \sigma_{\bar{a}} \models \varphi$. Note that $\Delta^{\Pi_1,\Pi_2}_{\mathcal{A},\bar{a}}$ is unique. For an ASM Π, $\Delta^{\Pi}_{\mathcal{A},\bar{a}}$ abbreviates $\Delta^{\Pi,\Pi}_{\mathcal{A},\bar{a}}$.

Since T_Π contains all formulae whose valuation influences a computation step of Π, the following lemma holds.

Lemma 15 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be states of an ASM $\Pi \in \mathcal{D}$, $\bar{a}$ be a tuple of elements from the domain of $\mathcal{A}_1$, $\bar{b}$ be a tuple of elements from the domain of $\mathcal{A}_2$ such that $\Delta^{\Pi}_{\mathcal{A}_1,\bar{a}} = \Delta^{\Pi}_{\mathcal{A}_2,\bar{b}}$. Furthermore, let $\mathcal{A}_1 \vdash_\Pi \mathcal{B}_1$ and $\mathcal{A}_2 \vdash_\Pi \mathcal{B}_2$. Then $\Delta^{\Pi}_{\mathcal{B}_1,\bar{a}} = \Delta^{\Pi}_{\mathcal{B}_2,\bar{b}}$.*

By induction on the number of steps, we obtain the following corollary.

Corollary 16 *Let $\mathcal{A}$ and $\mathcal{B}$ be states of an ASM $\Pi \in \mathcal{D}$, $\bar{a}$ be a tuple of elements from the domain of $\mathcal{A}$, $\bar{b}$ be a tuple of elements from the domain of $\mathcal{B}$ such that $\Delta^{\Pi}_{\mathcal{A},\bar{a}} = \Delta^{\Pi}_{\mathcal{B},\bar{b}}$. 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 $i \in \mathbb{N}$ $\Delta^{\Pi}_{\mathcal{A}_i,\bar{a}} = \Delta^{\Pi}_{\mathcal{B}_i,\bar{b}}$.*

The following lemma provides the possibility to conclude from standard states to arbitrary states.

Lemma 17 *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 $(R, \bar{a})$ where $R\bar{a}$ 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 $\bar{a}$ (where n is the number of variables occuring in T_{Π_1,Π_2}) of elements from the domain of $\mathcal{A}$, there is a type $\Delta_{\mathcal{A},\bar{a}}^{\Pi_1,\Pi_2} \in Q$ that contains exactly those formulae $\varphi \in T_{\Pi_1,\Pi_2}$ with $\mathcal{A}, \sigma_{\bar{a}} \models \varphi$ ($\sigma_{\bar{a}}$ is the variable assignment induced by $\bar{a}$, see above). According to the definition of $\mathcal{A}_Q$, there is an n-tuple $\bar{b}$ of elements from the domain of $\mathcal{A}_Q$ (the standard state for Q) such that $\mathcal{A}_Q, \sigma_{\bar{b}} \models \varphi$ holds for exactly those formulae $\varphi \in T_{\Pi_1,\Pi_2}$ that are contained in $\Delta_{\mathcal{A},\bar{a}}^{\Pi_1,\Pi_2}$. Consequently, $\bar{b}$ realizes $\Delta_{\mathcal{A},\bar{a}}^{\Pi_1,\Pi_2}$ in $\mathcal{A}_Q$.

Using corollary 16, we obtain that $\Delta_{\mathcal{A}_i,\bar{a}}^{\Pi_1} = \Delta_{\mathcal{A}_i^Q,\bar{b}}^{\Pi_1}$ and $\Delta_{\mathcal{B}_i,\bar{a}}^{\Pi_2} = \Delta_{\mathcal{A}_i^Q,\bar{b}}^{\Pi_2}$ for all $i \in \mathbb{N}$. Assuming that $\Pi_1 \equiv_S^{\mathcal{A}_Q} \Pi_2$, this implies that $\Delta_{\mathcal{A}_i,\bar{a}}^{\Pi_1,\Pi_2} = \Delta_{\mathcal{B}_i,\bar{a}}^{\Pi_1,\Pi_2}$ for all $i \in \mathbb{N}$.

As $\bar{a}$ 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 $\Pi_1 \equiv_S^{\mathcal{A}} \Pi_2$. $\square$

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}. In [1], it has been proven that a GF-formula Φ is satisfied by some state iff Φ holds in some quasistate. Using this fact and lemma 17, we obtain the following lemma.

Lemma 18 *Let $\Pi_1, \Pi_2 \in \mathrm{GF}(\mathcal{D})$. If $\Pi_1 \equiv_S^{\mathcal{A}_Q} \Pi_2$ holds for all $(T_{\Pi_1,\Pi_2} \cap \mathrm{GF})$-quasistates Q then $\Pi_1 \equiv_S \Pi_2$.*

Note that this is the point where we *need* to introduce the restriction to the guarded fragment. The reason is that the proposition from [1] does not hold for arbitrary FO-formulae but for formulae from GF.

7 Computing a Minimal Slice

In this section, we give the proof of theorem 9 stating that 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 of theorem 9. From lemma 18 we know the following. In order to check for two ASMs Π_1 and Π_2 from 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, we can assume a_Q to be $\prod_{R/n_R \in \Upsilon \text{ is a relation symbol}} 2^{|A_Q|^{n_R}}$ 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 $\mathcal{A}_Q = \mathcal{A}_0^Q \vdash_{\Pi_1} \mathcal{A}_1^Q \vdash_{\Pi_1} \mathcal{A}_2^Q \vdash_{\Pi_1} \cdots \vdash_{\Pi_1} \mathcal{A}_i^Q \vdash_{\Pi_1} \cdots \vdash_{\Pi_1} \mathcal{A}_{a_Q+1}^Q$ and $\mathcal{A} = \mathcal{B}_0 \vdash_{\Pi_2} \mathcal{B}_1 \vdash_{\Pi_2} \mathcal{B}_2 \vdash_{\Pi_2} \cdots \vdash_{\Pi_2} \mathcal{B}_i \vdash_{\Pi_2} \cdots \vdash_{\Pi_2} \mathcal{B}_{a_Q+1}^Q$. Now, check whether $\mathcal{A}_i|_{L(\mathcal{A}_i,S)} = \mathcal{B}_i|_{L(\mathcal{A}'_i,S)}$ for all $i \in \{1, \ldots, a_Q + 1\}$.

For two ASMs Π_1 and Π_2, the set of $(T_{\Pi_1,\Pi_2} \cap \text{GF})$-quasistates is finite. Therefore, it is decidable whether $\Pi_1 \equiv_S \Pi_2$ for $\Pi_1, \Pi_2 \in$ GF($\mathcal{D}$). This is a consequence of lemma 18.

As sub(Π) 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 $\Pi' \in$ sub(Π), check whether $\Pi \equiv_S \Pi'$. Return a minimal ASM from sub(Π) satisfying this condition. □

8 Complexity

From the proof of theorem 9 and the proofs of the involved lemmas, we can directly derive an algorithm computing a minimal slice. There exists a polynomial p such that this algorithm outputs a minimal after at most $2^{2^{2^{p(|\Pi|)}}}$ steps for any ASM $\Pi \in$ GF($\mathcal{D}$). Furthermore, we can easily reduce the finite satisfiability problem for GF to computing a minimal slice for ASMs from GF($\mathcal{D}$). This problem is 2EXPTIME-complete.

But 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 that 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 breaked off as soon as there is a difference in a run (relative to the slicing crtiterion) 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.

9 Extensions of the Basic Result

Theorem 9 can be extended rather easily in different ways. We give two examples.

Extension of $\mathcal{D}$ by Nondeterminism. By additionally allowing the choice of a tuple of elements from the domain of a state, we can extend the class $\mathcal{D}$ of ASMs to the class $\mathcal{ND}$.

The proof of the computability of a minimal slice for nondeterministic ASMs from GF($\mathcal{ND}$) can be done rather analogously to the one of theorem 9. 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.

Extension of $\mathcal{D}$ by import. By additionally allowing the use of **import**, we can extend the class $\mathcal{D}$ of ASMs to the class $\mathcal{DI}$.

W.l.o.g., we can assume that v appears only on the left-hand side of an update rule in a rule R appearing in the form **import** v R **endimport**.

The proof of the computabililty of a minimal slice for deterministic ASMs from GF($\mathcal{DI}$) proceeds similar to the one for deterministic ASMs from GF($\mathcal{D}$).

As in 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 reductions of them. 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 computation step, the set of imported elements is reduced to a representative set of limited size.

10 Variations in the Notion of Slicing

Dynamic and Static Slicing. In the literature, there are two basic versions of slicing, namely static and dynamic slicing. For dynamic slicing of a program, a specific input is given and we are only interested in a subprogram such that the behaviour is the same for *this* specific input. In static slicing, the input is not given. We are interested in the same behaviour for *all* possible inputs. Therefore, slicing in this work corresponds to static slicing.

For the classes $\mathcal{D}$ and $\mathcal{ND}$ (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 relative to a slicing criterion. Consequently, comparing the computation graph of an ASM Π and a given structure to the computation graphs of all elements of $\mathrm{sub}(\Pi)$ on this state leads to a minimal (dynamic) slice.

If the use of **import** is allowed then we can also compute a minimal (dynamic) slice assumed that we have only relation and constant symbols. The algorithm is similar to the one in the case without import but we do not consider the complete computation graph. We do not insert *all* states into the computation graph. If the difference between two states is only caused by importing different elements, then these states are identified and only one of them is inserted into the computation graph as node.

Limiting 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 ASM but only in the initial segments of length $n \in \mathbb{N}$ of all runs. This applies for example if we know that an error occurs during the first n steps of a computation.

Similar to the notion of S-slice, we can introduce the notion of (S, n)-slice which is a program from $\mathrm{sub}(\Pi)$ that, relative to S, behaves in the same way as Π on every initial segment of length n.

The proof of the computability of a minimal (S, n)-slice for $\mathrm{GF}(\mathcal{D})$ can be done completely analogously to the proof of theorem 9 except that we have to consider the runs on the standard states $\mathcal{A}_Q$ only up to the length n.

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 S-slice, we can introduce the notion of (S, φ)-slice which is a program from $\mathrm{sub}(\Pi)$ that, relative to S, behaves in the same way as Π on every run whose initial state satisfies φ.

The proof of the computability of a minimal conditioned slice for ASMs from $\mathrm{GF}(\mathcal{D})$ and properties from GF can be done completely analogously to the proof of theorem 9 execpt that we have to restrict to the quasistates satisfying φ.

Semantic Slicing. Instead of giving a slicing criterion as a set of atomic formulae, one could also consider an ASM Π and a property φ (given some formalism resp. logic) such that one is interested in a minimal subprogram Π' of Π that behaves exactly the same way as Π relative to φ.

Similar to the notion of S-slice, we can introduce the notion of φ-slice which is an element of $\mathrm{sub}(\Pi)$ that, relative to the valuation of φ, behaves in the same way as Π on every run.

The proof of the computability of a minimal semantic slice for ASMs from $\mathrm{GF}(\mathcal{D})$ and properties from GF can again be done similar to the one above with the following changes.

11 Conclusion and Future Work

We have presented an approach to transfer the notion of program slicing to ASMs. Although a minimal (static) slice is not computable in the general case, we have introduced an expressive class of ASMs for which a minimal (static) slice is computable. This basic computability result can be extended to larger classes of ASMs. Furthermore, we have demonstrated how it can be transfered to further notions of slicing to ASMs. These continuative considerations can be done rather easily once we have obtained the basic notion of static slicing for ASMs and the respective proof of the computability result.

Currently, we are implementing a system providing the possibility to slice ASMs from the presented class. This implementation is based on the algorithm derivable from the computability proof. There is a number of possibilities for optimizing the algorithm as we have already explained in the section on the complexity. We are aiming to obtain a variant which computes minimal slices within acceptable time for most cases.

Acknowledgements. I want to thank Erich Grädel for many helpful remarks on earlier drafts of this paper. I also want to thank the anonymous referees for their useful suggestions.

References

1. H. Andréka, J. van Benthem, and I. Németi. Modal Languages and Bounded Fragments of Predicate Logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.
2. A. Gargantini and E. Riccobene. ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science*, 7(11), 2001.
3. 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.
4. Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, EECS Dept, University of Michigan, 1997.
5. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.
6. M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.