TIK Report 136
The Formal Definition of Anlauff's eXtensible Abstract State Machines

Philipp W. Kutter[*]
Applied Formal Methods Institute AG and
Institute TIK, ETH Zürich

June 5, 2002

1 Introduction

*eXtensible Abstract State Machines (*XASM*)* [1, 4, 2, 3] has been designed and implemented by Anlauff as formal development tool for the Montages project. Unfortunately a formal semantics of XASM has not been given up to now. We streamline Anlauff's original design and present a denotational semantics, complementing the existing informal description. In fact we found that XASM implement a semantic generalization of Gurevich's *Abstract State Machines (ASMs)* [13, 14, 15, 16]. The initial idea for this generalization came from May's work [22] which is the first paper formalizing sequential composition, iteration and hierarchical structuring of ASMs. May notes that his approach complements

> .. the method of refining Evolving Algebras[1] by different abstraction levels [7]. There, the behavior of rules performing complex changes on data structures in abstract terms is specified on a lower level in less abstract rules, and the finer specification is proven to be equivalent. For execution, the coarser rule system is *replaced* by the finer one. In contrast, in the hierarchical concept presented here, rules specifying a behavior on a lower abstraction level are encapsulated as a system which is then *called* by the rules on the above level. [22], Section 6, page 14, 29ff

[*] Mailing address: Institute TIK, ETH Zentrum, Gloriastrasse 35, CH-8092 Zurich, Switzerland. phone: +41 1 260 75 52 email: kutter@tik.ee.ethz.ch

[1] Evolving Algebras is the previous name of ASMs.

XASM embeds this idea in the form of the "XASM call" into a realistic programming language design. The XASM call allows to model recursion in a very natural way, corresponding directly to recursive procedure calls in imperative programming languages. Arguments can be passed "by value", part of the state can be passed "by reference", the "result" of the call is returned as value allowing for functional composition, and finally the "effects" of the called machine are returned at once, maintaining the referential transparency property of non-hierarchical ASMs. Börger and Schmidt give a formal definition of a special case of the XASM call [8] where sequentiality, iteration, and parameterized, recursive ASM calls are supported. Unfortunately they are excluding the essential feature of both Anlauff's and May's original call to allow returning not only update sets, but as well a value. This restriction makes their call useless for the modeling of recursive algorithms.

The full XASM call leads to a design where every construct (including expressions and rules of Gurevich's ASMs) is denoted by both a value and an update set. This is a generalization of Gurevich's definition of ASMs, where the meaning of an expression is denoted by a value and the meaning of a rule is denoted by an update set [16].

For formalizing the full XASM call we are not completing the structural operational definitions given by Börger and Schmidt, but we extend Gurevich's denotational characterization of ASMs [16]. For completeness this denotational characterization of Gurevich's ASMs is repeated in Section 2. XASM are motivated and formalized in Section 3. The features of a pure functional sublanguage of XASM, including constructor terms, pattern matching, and derived functions are given in Section 4. Finally, in Section 5 we discuss related work.

2 Formal Semantics of ASMs

The mathematical model behind an ASMs is that a state is represented by an algebra or Tarski structure [27] i.e. a collection of functions and a universe of elements, and state transitions occur by updating functions point wise and creating new elements. Of course not all functions can be updated. The basic arithmetic operations (like add, which takes two operands) are typically not redefinable. The updatable or *dynamic functions* correspond to data-structures of imperative programming languages, while the static functions correspond to traditional mathematical functions whose definition does not depend on the current state. All functions are defined over the set $\mathfrak{U}$ of elements. In ASM parlance $\mathfrak{U}$ is called the *superuniverse*. This set always contains the distinct elements *true*, *false*, and *undef*. Apart from these $\mathfrak{U}$ can contain numbers, strings, and possibly anything – depending on what is being modeled. Subsets of the superuniverse $\mathfrak{U}$, called *universes*, are modeled by unary functions from $\mathfrak{U}$ to *true*, *false*. Such a function returns *true* for all elements belonging to the universe, and *false* otherwise. The universe *Boolean* consists of *true* and *false*. A function *f* from a universe *U* to a universe *V* is a unary operation on the superuniverse such that for all $a \in U$, $f(a) \in V \cup \{undef\}$ and *f(a)* = *undef* otherwise.

Functions from Cartesian products of $\mathfrak{U}$ to Boolean are called *relations*. By declaring a function as a relation, it is initialized for all arguments with *false*. A universe corresponds to a unary relation. Both universes and relations are special cases of functions. The dynamic functions not being relations are initially equal to *undef* for all arguments.

Formally, the *state* λ of an ASM is a mapping from a signature Σ (which is a collection of function symbols) to actual functions. We use f_λ to denote the function which corresponds to the symbol *f* in the state λ.

As mentioned above, the basic ASM *transition rule* is the update. An update rule is of the form

$$f(t_1, \ldots, t_n) := t_0$$

where $f(t_1, \ldots, t_n)$ and t_0 are closed terms (i.e. terms containing no free variables) in the signature Σ. The semantics of such an update rule is this: evaluate all the terms in the given state, and redefine the function corresponding to *f* at the value of the tuple resulting from evaluating $(t_1, \ldots, t_n)$ to the value obtained by evaluating t_0. Such a point wise redefinition of a function is called an *update*. Rules are composed in a parallel fashion, so the corresponding updates are all executed at once. Apart from the basic transition rule shown above, there also exist *conditional* rules, *do-for-all* rules, *choose* rules and lastly *extend* rules. Transition rules are recursively built up from these rules. The semantics of a rule is given by the set of updates resulting from composing updates of rule components. This so called *update denotation* of rules is formalized in the following.

Def. 1 Update denotation. *The formal semantics of a rule* R *in a state* λ *is given by its update denotation*

$$\mathit{Upd}(R, \lambda)$$

which is a set of updates.

The resulting state-transition changes the functions corresponding to the symbols in Σ in a point wise manner, by applying all updates in the set. The formal definition of an update is given as follows.

Def. 2 Update. *An update is a triple*

$$(f, (e_1, \ldots, e_n), e_0)$$

where f *is a* n*-ary function symbol in* Σ *and* $e_0, \ldots, e_n$ *are elements of* $\mathfrak{U}$*.*

Intuitively, firing this update in a state λ changes the function associated with the symbol *f* in λ at the point $(e_1, \ldots, e_n)$ to the value e_0, leaving the rest of the function (i.e. its values at all other points) unchanged. Firing a rule is done by firing all updates in its update denotation.

Def. 3 Successor state. *Firing the updates in Upd*(R, λ_i) *in the state* λ_i *results in its successor state* λ_{i+1}. *For any function symbol f from* Σ, *the relation between* f_{λ_i} *and* $f_{\lambda_{i+1}}$ *is given by*

$$f_{\lambda_{i+1}}(e_1, \ldots, e_n) \; = \; \begin{cases} e_0 \quad if \quad (f, (e_1, \ldots, e_n), e_0) \in \textit{Upd}(R, \lambda_i) \\ f_{\lambda_i}(e_1, \ldots, e_n) \quad otherwise \end{cases}$$

There are two remarks concerning this definition. First, if there are two updates which redefine the same function at the same point to different values, the resulting equations are inconsistent, and the next state $f_{\lambda_{i+1}}$ cannot be calculated. Consistency of rules cannot be guaranteed in general, and an inconsistent rule results in a system abort.

The second remark is about completeness of the successor-state relation. The above complete definition of the next state (Def. 3) could be relaxed to a partial definition as follows:

Def. 4 Partial successor state *Firing the updates in Upd*(R, λ_i) *in the state* λ_i *results in its successor state* λ_{i+1}. *For any function symbol* f *from* Σ, *the relation between* f_{λ_i} *and* $f_{\lambda_{i+1}}$ *must be a model for the following equations:*

$$f_{\lambda_{i+1}}(e_1, \ldots, e_n) \; = \; e_0 \quad if \quad (f, (e_1, \ldots, e_n), e_0) \in \textit{Upd}(R, \lambda_i)$$

The advantage of the partial definition becomes visible in approaches where ASM rules are modeled as equation systems, for instance if ASMs are modeled with Algebraic Specifications [20, 21, 25, 26]. The complete definition results in an exploding number of equations [20, 21] while the partial definition allows to solve this problem elegantly [26]. Further the partial definition Def. 4 allows to *compose* the equations of the subrules, whereas the complete definition does not allow for such a composition.

The different forms of rules are given below. We use $eval_\lambda$ to denote the usual term evaluation in the state λ. In all definitions, $t_0, \ldots, t_n$ are terms over Σ.

Def. 5 Update denotations of ASM rules.

Basic Update

if $R = f(t_1, \ldots, t_n) := t_0$
then $\quad$ *Upd*$(R, \lambda) = (f, (eval_\lambda(t_1), \ldots, eval_\lambda(t_n)), eval_\lambda(t_0))$

Parallel Composition

if $R = R_1 \ldots R_m$
then $\quad$ *Upd*$(R, \lambda) = \bigcup_{i \in \{1, \ldots, m\}}$ *Upd*(R_i, λ)

Conditional Rules

if R = **if** t **then** R_{true} **else** R_{false} **endif**
then $\quad$ *Upd*$(R, \lambda) = \begin{cases} \textit{Upd}(R_{true}, \lambda) \quad if \quad eval_\lambda(t) = true \\ \textit{Upd}(R_{false}, \lambda) \quad otherwise \end{cases}$

Do-for-all

```
if R = do forall x in U : Q(x)
          R'
        enddo
then    $Upd(R, \lambda) = \bigcup_{e \in U'} Upd(R', \lambda \cup \{x \mapsto e\})$
where
```

- $U' = \{e|\ eval_\lambda(U(e) \wedge Q(e))\}$ *are U elements fulfilling Q.*
- $\lambda \cup \{x \mapsto e\}$ *is state* λ *with x interpreted as e.*

Choose

```
if R = choose x in U : Q(x)
          R'
        ifnone
          R"
        endchoose
then
```

$$Upd(R, \lambda) = \begin{cases} Upd(R', \lambda \cup \{x \mapsto ORACLE\}) & \\ & if \quad \exists e : eval_\lambda(U(e) \wedge Q(e)) \\ Upd(R'', \lambda) & otherwise \end{cases}$$

where ORACLE is a nondeterministically chosen element $e \in U_\lambda$ [2]*, fulfilling Q(e).*

Extend

```
if R = extend U with x
         R'
       endextend
then    $Upd(R, \lambda) = Upd(R', \lambda_{x \mapsto e}) \cup \{(U, (e), true)\}$,
```

where e does not belong to the domain or the co-domain of any of the functions in state λ*, i.e. is a new, unused element.*

3 The XASM Specification Language

Due to the fact that the ASM approach defines a notion of *executing* specifications, it provides a perfect basis for a language, which can be used as a specification language as well as a high-level programming language. However, in order to upgrade to a realistic programming language, such a language must – besides other features – add a modularization concept to the core ASM constructs in order to provide the possibility to structure large-scale ASM-formalizations and to flexibly define reusable specification units. XASM realizes a component-based modularization concept based

[2] As mentioned, U_λ is the definition of U in state λ.

on a unification and generalization of ASM's rule and expression semantics. The unification of rules and expressions is done by considering each ASM construct, whether rule or expression, to have both an update set denotation, and to evaluate to a result, the so called *value denotation*.

In addition to the existing ASM constructs, we introduce a new feature, so called *external functions*[3] External functions can be evaluated like normal functions, but as a result, both a value, and an update set is returned. They correspond to procedures known from imperative programming languages. For each external function, we need to specify its *update denotation* and its *value denotation*. Both denotations can be freely defined. The formal definition of external functions, their denotations, and the propagation of these denotations through the existing ASM term and rule constructors is given in Section 3.1.

While external functions make the calculation of rule sets, and thus the semantics of XASM rules *extensible,* we introduce a second new construct called *environment functions* in order to make XASM*open* to the outside computations. *Environment functions* are special dynamic functions whose initial definition is given as a parameter to an ASM. After an ASM terminates, the aggregated updates of the environment functions are returned as update denotation of the a complete ASM run. The formalization of ASM runs, environment functions, the update denotation of an ASM run in terms of state-delta, and the value denotation of an ASM run are given in Section 3.2.

For intuition, it is a good idea to think about environment functions as dynamic-functions passed to an ASM as reference parameters, and about external functions as locally declared procedures. Having both concepts we can plug the two mechanisms together by defining update and value denotation of an external function by means of an ASM run. Thus the evaluation of such an external function corresponds to running, or *calling* another ASM. The environment functions of the called ASM are given as functions of the calling ASM. The details how an external function can be realized as ASM are given in Section 3.3. The formalization is given by using the definition of update and value denotations of an ASM, as defined in Section 3.2 as the definition of the update and value denotations of the realized external function.

3.1 External Functions

In Section 2 the denotation of each ASM rule construct has been given as a set of updates. Denotation of terms has been formalized by means of the usual $eval_{\lambda}$ term evaluation. The denotation of each existing ASM construct is thus either a set of updates or an element, the result of its evaluation. The ASM constructs denoted by updates are the *rules*, and the ASM constructs denoted by values are the *terms*.

The idea of eXtensible ASMs (XASM) is to unify rules and terms, by considering each construct to have both an update and a value denotation. In pure ASMs rules would have the value denotation *undef* and expressions have the empty set as update

[3] In the context of ASMs the term ”external function” has been used in a slightly different way. For the sake of simplicity we are using the term ”external function” only in connection with XASM, and not with ASMs and we are always referring to the XASM definition of ”external function”.

denotation. In XASM*external functions* are introduced as a new construct having both denotations.

In order to avoid confusion with the standard $eval_{\lambda}$ function, we introduce a new function which gives the value-denotation.

Def. 6 Value denotation. *The value denotation of each rule or expression* R *in a state* λ *is given by*

$$\text{Eval}(R, \lambda)$$

which is an element of $\mathfrak{U}$.

The external functions are declared using the keyword *external function*. Syntactically the external functions are used like normal functions. Function composition which involves external functions may thus result in updates, and we need therefore to redefine the update denotations of all rule constructions involving expressions, by refining Def. 5.

In order to simplify the presentation of semantics, we denote the external function symbols with underlined symbols, for instance $\underline{\mathrm{f}}$. These symbols are grouped in the set Σ_{ext} of external symbols.

Def. 7 Extended signature. *The signature* Σ *is extended with the symbols* Σ_{ext} *of* external functions *to signature* Σ'.

$$\Sigma' = \Sigma \cup \Sigma_{ext}$$

Since the external functions are not part of an ASM's state, the definition of state λ is not affected, it is still a mapping from signature Σ of dynamic functions to the actual definitions of these functions. However, terms can be built over the extended signature Σ'.

Def. 8 Denotations of external functions. *For each external function* $\underline{f} \in \Sigma_{ext}$ *we must define the functions*

$$\text{ExtUpd}(\underline{f}, (e_1, \ldots, e_n), \lambda)$$

and

$$\text{ExtEval}(\underline{f}, (e_1, \ldots, e_n), \lambda)$$

which are calculating in state λ *the updates and value denotations of the external function* $\underline{f}$ *given parameters* $(e_1, \ldots, e_n)$.

XASM features interfaces allowing to give these definitions in arbitrary external languages, which leads to a non-formal system, or in XASM itself, which leads to a formal system which is described in Section 3.3.

In the following we give the definition of *Upd* and *Eval* for function composition of dynamic functions $f \in \Sigma$, external functions $\underline{f} \in \Sigma_{ext}$, and all six rule constructors.

Def. 9 Update and value denotations of XASM constructs. *Assume in all following definitions*

- $t_0, \ldots, t_n$ *are terms over* Σ',
- $e_0 = \mathit{Eval}(t_0, \lambda)$ *and* $\ldots$ *and* $e_n = \mathit{Eval}(t_n, \lambda)$ *are the elements these terms evaluate to,*
- $f \in \Sigma$ *is the symbol of a dynamic function, and*
- $\underline{f} \in \Sigma_{ext}$ *is the symbol of an external function.*

Function Composition

if $R = f(t_1, \ldots, t_n)$
then
$\mathit{Upd}(R, \lambda) = \bigcup_{i \in \{1,\ldots,n\}} Upd(t_i, \lambda)$
$\mathit{Eval}(R, \lambda) = f_\lambda(e_0, \ldots, e_n)$

External Function Composition

if $R = \underline{f}(t_1, \ldots, t_n)$
then
$\mathit{Upd}(R, \lambda) = \mathit{ExtUpd}(\underline{f}, (e_0, \ldots, e_n), \lambda) \quad \cup \quad \bigcup_{i \in \{1,\ldots,n\}} Upd(t_i, \lambda)$
$\mathit{Eval}(R, \lambda) = \mathit{ExtEval}(\underline{f}, (e_0, \ldots, e_n), \lambda)$

Basic Update

if $R = f(t_1, \ldots, t_n) := t_0$
then
$\mathit{Upd}(R, \lambda) = \{(f, (e_1, \ldots, e_n), e_0)\} \quad \cup \quad \bigcup_{i \in \{1,\ldots,n\}} Upd(t_i, \lambda)$
$\mathit{Eval}(R, \lambda) = \mathit{undef}$

Conditional Rules

if R = **if** t **then** R_{true} **else** R_{false} **endif**
then

$$\mathit{Upd}(R, \lambda) = \begin{cases} \mathit{Upd}(R_{true}, \lambda) \ \cup \ \mathit{Upd}(t, \lambda) & if \quad \mathit{Eval}(t, \lambda) = true \\ \mathit{Upd}(R_{false}, \lambda) \ \cup \ \mathit{Upd}(t, \lambda) & otherwise \end{cases}$$

$$\mathit{Eval}(R, \lambda) = \begin{cases} \mathit{Eval}(R_{true}, \lambda) & if \quad \mathit{Eval}(t, \lambda) = true \\ \mathit{Eval}(R_{false}, \lambda) & otherwise \end{cases}$$

Parallel Composition

if $R = R_1 \ldots R_m$
then
$\mathit{Upd}(R, \lambda) = \bigcup_{i \in \{1,\ldots,m\}} \mathit{Upd}(R_i, \lambda)$
$\mathit{Eval}(R, \lambda) = undef$

Do-for-all

```
if R = do forall x in U : Q(x)
          R'
        enddo
then
```

$$\begin{aligned} \mathit{Upd}(R,\lambda) = & \bigcup_{e1 \in U'} \mathit{Upd}(R', \lambda \cup \{x \mapsto e1\}) \\ & \cup \quad \bigcup_{e2 \in U_\lambda} Upd(Q(e2),\lambda) \\ \mathit{Eval}(R,\lambda) = & \ undef \end{aligned}$$

where

- $U' = \{e| \mathit{Eval}(U(e) \wedge Q(e), \lambda)\}$ *are U elements fulfilling Q.*
- $\lambda \cup \{x \mapsto e\}$ *is state* λ *with x interpreted as e.*

Choose

```
if R = choose x in U : Q(x)
          R'
        ifnone
          R"
        endchoose
then
```

$$\mathit{Upd}(R,\lambda) = \begin{cases} \mathit{Upd}(R', \lambda \cup \{x \mapsto \mathit{ORACLE}\}) \\ \quad \cup \ \mathit{Upd}(Q(\mathit{ORACLE}), \lambda) \\ \qquad\qquad\quad if \quad \exists e : \mathit{Eval}(U(e) \wedge Q(e), \lambda) \\ \mathit{Upd}(R'', \lambda) \\ \qquad\qquad\quad otherwise \end{cases}$$

$$\mathit{Eval}(R,\lambda) = \begin{cases} \mathit{Eval}(R', \lambda \cup \{x \mapsto \mathit{ORACLE}\}) \\ \qquad\qquad\quad if \quad \exists e : \mathit{Eval}(U(e) \wedge Q(e), \lambda) \\ \mathit{Eval}(R'', \lambda) \\ \qquad\qquad\quad otherwise \end{cases}$$

where

ORACLE is a nondeterministically chosen element $e \in U$*,*
fulfilling Q(e) in λ*.*

Extend

```
if R = extend U with x
          R'
        endextend
then
```

$$\mathit{Upd}(R,\lambda) = \mathit{Upd}(R', \lambda \cup \{x \mapsto e\}) \cup \{(U, (e), true)\},$$

$$\mathit{Eval}(R,\lambda) = undef$$

where

e does not belong to the domain or the co-domain of any of the functions in state λ.

3.2 Semantics of ASM run and Environment Functions

We have given the semantics of ASM rules and expressions in terms of defining the relation of one state to the next. In this section we formalize how the state of an ASM is initialized, by means of *parameters* and so called *environment functions*, and what is an ASM run. We give both value and update denotations of ASM runs.

We mentioned earlier that dynamic functions are initialized everywhere with undef, except for relations, which are initialized everywhere with false. Parameters and environment functions allow to initialize functions with different values. As example we take the following ASM.

ASM 1 `asm InitializationExample(p1, p2)`

```
  updates function  f(_,_)
  accesses function g(_,_)
is
 function h(_,_)
R
endasm
```

The example shows two parameters, *p1* and *p2*, two environment functions, *f* and *g*, and one normal dynamic function *h*. If the ASM is started, or *called*, actual values for the parameters have to be given, as well as definitions for the environment functions. Parameters result in normal, 0-ary dynamic functions, which are initialized with the actual value. Environment functions are used to initialize functions of arity higher than zero. As we can see, there are two ways to declare environment functions, one for read-only access as ”accesses” and the other for read-write access as ”updates”. In addition to such declared functions there is the special 0-ary function *result* which is used to return values from an ASM run.

Intuitively environment functions correspond to reference parameters passed to an ASM call. The aggregated updates to these functions constitute the update denotation of an ASM run. In contrast parameters can be considered call-by-value arguments. Updates to such arguments are possible in Xasm, but they have only local effects.

The signature Σ of the state of an ASM consists thus of the normal dynamic functions, the 0-ary dynamic functions initialized by actual parameters, the environment functions, and the special function *result*.

Def. 10 Local and environment functions. *The signature Σ of dynamic functions is built by a set of locally defined functions Σ_{loc}, the set of parameter functions Σ_{par}, the set of environment functions Σ_{env} and the special function* result. *All of them must be* *disjoint.*

$$\begin{array}{rcl} \Sigma & = & \Sigma_{env} \cup \Sigma_{loc} \cup \Sigma_{par} \cup \{result\} \\ & \wedge & \\ \Sigma_{env} \cap \Sigma_{loc} \cap \Sigma_{par} \cap \{result\} & = & \{\} \end{array}$$

An ASM can now be called by providing it with actual parameters, and an initial state for the environment functions.

Def. 11 ASM call. *An ASM with rule R parameters $p_1, \ldots, p_n$ and environment functions Σ_{env} is called by the following triple*

$$(R, (a_1, ..., a_n), \lambda^e)$$

where $(a_1, \ldots, a_n)$ are actual values for the parameters of the ASM, and λ^e is a mapping from the function symbols of Σ_{env} to actual definitions for these functions.

Given an ASM call, we can define the initial state of the called ASM as follows.

Def. 12 Initial state. *Given an ASM call $(R, (a_1, ..., a_n), \lambda^e)$ with parameters $p_1, \ldots,$ p_n the initial state λ_0 of the called ASM is defined as follows.*

$$\lambda_0 = \lambda^e \cup \{p_0 \mapsto e_0, \ldots, p_n \mapsto e_n\}$$

Given the definition of the initial state and of the next state relation we can define the fixpoint semantics of an ASM run as follows.

Def. 13 Fixpoint semantics. *Given an ASM call $(R, (a_1, ..., a_n), \lambda^e)$, the definition of the initial state λ_0 of such a call, according to Def. 12, and the relation of state λ_i to λ_{i+1}, according to Def. 3, we define the fixpoint semantics Λ as a mapping from ASM calls to final states or $\perp$ if there is no fixpoint.*

$$\Lambda(R, (a_1, ..., a_n), \lambda^e) = \begin{cases} \lambda_i & if \quad \lambda_i = \lambda_{i+1}, not(\exists j : j < i : \lambda_j = \lambda_{j+1}) \\ \perp & if \quad not(\exists i :: \lambda_i = \lambda_{i+1}) \end{cases}$$

where $\perp$ denotes a non-terminating call.

Given the fixpoint semantics of an ASM call, we can define the update and value denotation of such a call. The value denotation is simply the value of function *result* in the final state of the call.

Def. 14 Value denotation of ASM call. *Given an ASM call $(R, (a_1, ..., a_n), \lambda^e)$, and the fixpoint semantics, according to Def. 13, the value denotation* CallEval *is the value of* result *in the final state of the call.*

$$\mathit{CallEval}(R, (a_1, ..., a_n), \lambda^e) = \mathit{result}_{\Lambda(R,(a_1,...,a_n),\lambda^e)}$$

The update denotation *CallUpd* of a call is given by the aggregated updates to environment functions. The aggregated updates are calculated by comparing the initial state and the terminal state of these functions. The comparison of states is done by *state subtraction*

Def. 15 State subtraction *Given two states λ_1 and λ_2 over the same signature Σ, the formal definition of state subtraction is*

$$\begin{array}{rcl} \lambda_1 - \lambda_2 & = & \{(f,(e_1,\ldots,e_n),e_0)| \\ & & \quad f \in \Sigma \wedge e_0,\ldots,e_n \in \mathfrak{U} \\ & & \quad \wedge\ f_{\lambda_1}(e_1,\ldots,e_n) = e_0 \\ & & \quad \wedge\ f_{\lambda_2}(e_1,\ldots,e_n) \neq e_0 \\ & & \} \end{array}$$

Using this definition, the update denotation of an ASM call is defined as follows.

Def. 16 Update denotation of ASM call. *Given an ASM call $(R,(a_1,...,a_n),\lambda^e)$, the signature Σ_{env} of environment functions, the fixpoint semantics, according to Def. 13, and the definition of state subtraction according to Def. 15, the update denotation* CallUpd *is the environment part of the final state minus the initial state λ^e of the environment functions.*

$$\textit{CallUpd}(R,(a_1,...,a_n),\lambda^e) = \Lambda(R,(a_1,...,a_n),\lambda^e)|_{\Sigma_{env}} - \lambda^e$$

3.3 Realizing External Functions with ASMs

After we specified both external functions, for which we need to give value and update denotations *ExtEval* and *ExtUpd*, and as well ASM calls, for which we defined value and update denotations *CallEval* and *CallUpd*, the next natural thing to do is to use the denotations of an ASM call as definitions of the denotations of an external function. With other words, we realize an external function with an ASM. The environment functions of the called ASM are naturally taken from the dynamic functions of the called ASM, and the resulting updates to these functions fit thus naturally in the update set of the calling ASM.

The definition of update set and value denotations of an external function realized by ASM can now be given by using *CallUpd* and *CallEval* as definitions of *ExtUpd* and *ExtEval*.

Def. 17 Denotations of ASM call. *Assume the external function _f to be implemented by the following ASM:*

```
asm _f(p1, ..., pn)
 updates functions SIGMA_ENV
is
functions SIGMA_LOC
 R
endasm
```

where SIGMA_ENV *is the signature* Σ^c_{env} *of environment functions of the called ASM, and* SIGMA_LOC *is the signature* Σ^c_{loc} *of locally declared dynamic functions of the called ASM..*

Given a state λ *of the ASM calling* $\underline{f}$*, the denotations* ExtUpd *and* ExtEval *are defined as follows.*

$$\begin{array}{rcl} \textit{ExtUpd}(\underline{f}, (e_1, \ldots, e_n), \lambda) & = & \textit{CallUpd}(\underline{f}, (e_1, \ldots, e_n), \lambda|_{\Sigma_{env}}) \\ \textit{ExtEval}(\underline{f}, (e_1, \ldots, e_n), \lambda) & = & \textit{CallEval}(\underline{f}, (e_1, \ldots, e_n), \lambda|_{\Sigma_{env}}) \end{array}$$

Examples Consider our previous example the ASM *AP*. An ASM *AQ*, can refer to *AP*, by declaring it as *external "ASM" function*, or short *external function*.

ASM 2
```
asm AQ is
    function i <- 0
    external function AP
  if i < 10 then
     i := i+1
     AP
  endif
endasm
```

In *AQ* there is a local 0-ary function *i*, and the external function *AP*, which is realized as ASM. The if-clause in the rule of *AQ* guarantees that *AP* is called 10 times. Each time, *AP* is called, it runs until its termination, the final state of *AP* is interpreted as an update set, and the value of the function *return* in *AP* is used as return value. The update set generated by each run of *AP* is

$$\{(x_1, (), 1), (x_2, (), 1), (x_3, (), 1)\}$$

Since all of the updated functions are local to *AP*, the generated update set has no effects on the state of *AQ*. Further, in this simple case, the value of *return* is *undef*, since there is no update to *return* in *AP*. Thus the value denotation of calling *AP* is *undef*.

As second example consider two ASMs *A* and *B*. We abstract from concrete rules and consider *A* to execute the parallel composition of a rule *Ra* and a call to *B*, while *B* is considered to execute a rule *Rb*. *A* has locally defined functions $a1, \ldots, a_n$ and *B* has locally defined functions $b_1, \ldots, b_m$.

ASM 3
```
asm A is
    functions a1, ...., an
    external function B
  Ra
  B
endasm
```

ASM 4
```
asm B
    updates functions a1, ..., an
```

```
is
    functions b1, ..., bm
  Rb
endasm
```

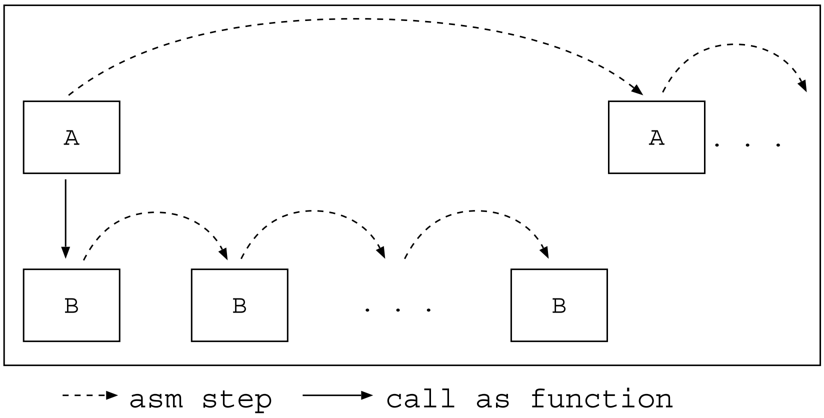

Figure 1: ASM A calls ASM B

The interface of *B* determines that ASM calling *B* must provide dynamic functions $a1, \ldots, a_n$ which are allowed to be updated by *B*.

The situation of *A* calling *B* is visualized in Fig. 1. In each step of *A*, the rule *Ra* as well as ASM *B* are executed. If *B* is called, the current state of *A*'s functions is passed to *B* as the initial state of the environment functions. From this state, *B* runs until its termination, updating the state of its local functions as well as the state of the environment functions. After termination, the state of the local functions of *B*, is discarded, and the state of the environment functions is compared with their initial state, passed by the environment. The changes with respect to the initial state are returned as the update-denotation of the *B*-call.

The updated-denotation of the *B*-call is combined with the update-denotation of the *Ra*-rule, and applied to the current state of *A*. Only now *A*'s locally defined functions are really updated. The internal steps of *B* are not visible to *A*. From *A*'s perspective, calling *B* is considered an atomic action. The XASM call provides thus an abstraction from sequentiality.

Returning values We have mentioned several times the special role of the function *result*, but we have not shown its use and examples. Based on the above definitions, *result* must be declared as local function and updated like any other function. The termination of an ASM does not a priori depend on the state of *result*. A typical "factorial"-program would look as follows.

ASM 5 `asm factorial(n) is`

```
    function result
  if result != undef then
    if n = 0 then
      result := 0
    else
      result := n * factorial(n-1)
    endif
  endif
endasm
```

For convenience a shorthand notation allows the user to skip the explicit declaration of the variable ”result”, as well as the outer ”if result != undef”-clause, and it introduces the more intuitive syntax ”return x” instead of ”result := x”. Applied to the previous example, the shorthand notation results into the following formulation.

ASM 6
```
asm factorial(n) is
  if n = 0 then
    return 0
  else
    return n * factorial(n-1)
  endif
endasm
```

As a last example of this section, we would like to show a formulation of “factorial” which avoids call-recursion.

ASM 7
```
asm factorial(n) is
    function n0 <- n, r <- 1
  if n0 > 0 then
    r := n0 * r
    n0  := n0 - 1
  else
    return r
  endif
endasm
```

Every tail-recursive algorithm can be reformulated in this iterative style. We will use this stile throughout the thesis, since it shows clearer how ASMs work. In the following variant of factorial we use the fact, that the parameters of an ASM can be used as normal 0-ary dynamic functions.

ASM 8
```
asm factorial(n) is
    function r <- 1
  if n > 0 then
    r := n * r
    n  := n - 1
  else
    return r
  endif
endasm
```

4 Constructors, Pattern Matching, and Derived Functions

Most theoretical case studies using ASMs start with a mathematical model of some static system, formalized as a fixed set of statically defined functions and elements, and add a number of dynamic functions on top of this algebra. With the up to now discussed features, the static models must be either provided by an external implementation, or simulated with dynamic functions as well.

While experimenting with early versions of XASM, we identified one mathematical concept which is on one hand often used and on the other hand very awkwardly simulated with dynamic functions. The identified concept is *free-generated-terms*. Unlike terms over dynamic functions, evaluating initially all to the same element *undef*, free-generated-terms, or *constructors* are expected to map to the same element, if and only if all their arguments are equal. This concept corresponds to free-data-types in functional programming languages like Standard ML [23, 9] or term algebras in algebraic specifications [12]. XASM features an untyped variant of classical constructor terms, as well as pattern matching and derived functions. These three features form a pure functional subset of XASM, but can as well be mixed freely with other features of XASM.

The definition of the two constructors

```
constructors zero, successor(_)
```

is thus not only creating the elements $\{$*zero, successor(zero), successor(successor(zero),* $\ldots\}$, but as well unexpected elements like *successor(true)* or *successor(*e_0*)*, where e_0 is an element created by an *extend*-rule; such elements do not correspond to any built-in constant symbol, XASM allows the user to define constructor-terms having no syntactical representation.

4.1 Pattern Matching

In combination with constructors, it is very useful to have *pattern matching* and *derived functions*. As an example for pattern matching, consider an abstract-data-type stack, being specified by the following equations.

$$\begin{array}{rcl} pop(push(s,v) & = & s \\ top(push(s,v) & = & v \\ top(empty) & = & undef \\ pop(empty) & = & empty \end{array}$$

Two constructors *empty* and *push(_, _)* are used to build stacks in the usual way. *top(_)* and *pop(_)* are declared as external functions and realized as ASMs. Within these ASMs, pattern matching is used.

ASM 9
```
constructors empty, push(_,_)
external functions top(_), pop(_)

asm top(s)
  accesses function push(_,_)
is
  if s =~ push(&, &v) then
    return &v
  else
    return undef
  endif
endasm

asm pop
  accesses functions empty, push(_,_)
is
  if s =~ push(&s, &) then
    return &s
  else
    return empty
  endif
endasm
```

We see the pattern matching symbol “=~” and the pattern variables, which all start with the symbol &. The plain symbol & is a placeholder for pattern variables, whose value is not used anymore. The matching-expression is given as condition of an if-then-else rule. If a match happens, the pattern-variables can be used, otherwise they cannot. Thus pattern-variables can only be used in the then-clause of an if-then-else rule.

4.2 Derived Functions

A third construct which is useful in combination with constructors and pattern-matching is the *derived function*. The value of derived function is defined by an expression. The derived function

```
derived function f(p1, ..., pn) == t
```

where *t* is a term build over Σ and the parameters $p_1, \ldots, p_n$, is semantically equivalent to an external function defined as follows.

```
external function f(p1, ..., pn)

asm  f(p1, ..., pn)
  accesses ...
is
  return t
endasm
```

Using derived functions, the above example ASM 9 can be reformulated as follows:

ASM 10
```
constructors empty, push(_,_)
derived function top(s) ==
  (if s =~ push(&, &v) then &v else undef)
derived function pop(s) ==
  (if s =~ push(&s, &) then &s else empty)
```

4.3 Formal Semantics of Constructors

The concept of terms built up by constructors can be mapped to the ASM approach as follows: each of the function names may be marked as *constructive*, expressing that constructor functions are one-to-one and total.

Let $\Sigma_c \subseteq \Sigma$ be the set of all constructive function symbols. If $f \in \Sigma_c$, be of arity *n*, $g \in \Sigma_c$, be of arity *m*, and $t_1, \ldots t_n, s_1, \ldots, s_m$ be terms over Σ', then the following condition hold for all states λ_i of the ASM:

(i)
$$f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$$

iff
$$(f = g) \text{ and } (n = m) \text{ and } (eval_{\lambda_i}(t_k) = eval_{\lambda_i}(s_k))$$

for all $1 \leq k \leq n$ where $eval_{\lambda_i}(t)$ stands for the evaluation of the term *t* in state λ_i of the ASM. Informally speaking it means that each constructive function is total with respect to $\mathfrak{U}$ and injective.

(ii) For all $j > i, e_1 \in \mathfrak{U}, \ldots, e_n \in \mathfrak{U}$

$$f_{\lambda_i}(e_1, \ldots, e_n) = f_{\lambda_j}(e_1, \ldots, e_n)$$

This means that constructive functions do not change their values with time, but whenever a new element is created, the domain of all constructive functions is automatically extended to the new element; from that moment on, all elements constructed from the newly defined element do not change in time either.

If $f \in \Sigma_c$, then *f* is called a *constructor*, and terms $f(t_1, \ldots, t_n)$ built only over Σ_c are called *constructor terms*. In the following, we use the constructor term *t* as a synonym for its unique value $eval_{\lambda}(t)$.

5 Related Work and Results

Gurevich's initial program for ASMs is pure mathematical: a mathematically defined dynamic system, which allows one to model arbitrary algorithms. His thesis [13] is that unlike Turing machines [28] his machines would allow to model algorithm without encoding data-structures and splitting execution steps. He observed that every conceivable data-structure can be modeled as a Tarski structure, and every possible state change of the algorithm can be modeled by a set of explicit, pointwise changes to the structure. In [17, 18] a proof of the thesis for sequential algorithms is given, and in [5] the corresponding proof for parallel algorithms is shown.

This pure mathematical program, has been implicitly transformed in a computer science project, by defining a concrete rule-language for defining the update sets. While in [13, 14] Gurevich is investigating the concept of dynamically changeable Tarski structures, later in [15] he defines a set of fixed, minimal languages for defining rules. ASMs are defined to correspond to this rule-programming-languages, and under this interpretation the thesis has subsequently provoked a lot of polarization among computer scientists. The lack of modularization and reuse feature in the proposed languages is for computer scientists not compatible with the claim, that arbitrary algorithms can be modeled on their natural abstraction level. While the initial mathematical meaning of this sentence makes a lot of sense, it contradicts computer scientist's experience, if "algorithm" is interpreted as software or hardware, and "modeled" is interpreted as "prototyped" or even "implemented" in a feasible and maintainable way.

However, the debate on ASMs in computer science has led to an impressive collection of case studies, each of them using ASMs to model a system which is considered to be complex. Examples are referenced in the annotated bibliography [6]. While most models try to restrict the used rule-languages to the predefined ones, in many cases additional machinery has been used in order to manage the complexity. Such machinery reuses common concepts from programming languages.

The functional programming paradigm has been considered as the best candidate for extending the minimal rule-languages. The reason is, that many theoretical ASM case studies use a considerable amount of higher mathematics to describe the static part of algorithms. Functional programming is ideal to model higher mathematics and it uses modularization concepts based on mathematical concepts. This approach has led to a number of ASM implementations based on functional languages [29, 11]. Odersky [24] proposes the opposite way, e.g. to use variable functions as an additional construct in functional programming languages. In both cases a functional type system is proposed. The introduction of such a type system is helpful for cases where the described algorithms fits well into the type system. On the other hand, Gurevich's original untyped definition of ASMs still provides the highest level of flexibility. We do not know of an ASM implementation based on functional languages which provides an implementation of the original, untyped definition of ASMs.

Today's software systems reached a level of complexity leading to use of multiple paradigms [10]. Our experience shows that untyped ASMs are useful to use different paradigms in parallel. The idea behind XASM is to start with Gurevich's untyped definition of ASMs in [14] and to make it extensible. The exact mechanisms have been discussed before. Unlike other extensions ASMs, the XASM approach does not alter the semantics idea of Tarski structures and update sets. The only difference of XASM to Gurevich's ASMs is, that we allow extensible rule languages. Since the means for extension are again ASMs, the XASM call can be seen as well as a way to structure ASMs.

An algebraic view of a similar structuring concept has been given by May in [22]. The XASM call is a special case of notions defined in [22]. While May applies the state of the art in algebraic specification technologies to ASMs, the idea of XASM is to generalize the original idea of Gurevich, resulting in a more practical specification and implementation tool. Unlike many other proposals for extending ASMs, the XASM approach tries to follow Gurevich's style to introduce as few concepts as possible. In fact, the XASM call is the only new concept and can be used to define all other extensions.

Agreeing on the choice of Tarski structures and update set for modeling algorithms, we claim that the current choice of ASM constructs is possibly not enough to fulfill the ASM thesis. Although theoretically every update set can be denoted by an appropriate ASM rule, the abstraction level how the update set is calculated is fixed. We propose to consider to remedy this problem by extending ASM such that the update sets can be calculated by means of another ASM.

References

[1] M. Anlauff. Aslan - programming in abstract state machines. A small stand–alone ASM interpreter written in C, ftp://ftp.first.gmd.de/pub/gemmex/Aslan.

[2] M. Anlauff. XASM - An Extensible, Component-Based Abstract State Machines Language. In Gurevich et al. [19], pages 69–90.

[3] M. Anlauff and P. W. Kutter. The xasm open source project. http://www.xasm.org, 2002.

[4] M. Anlauff, P. W. Kutter, and A. Pierantonio. Aslan: Programming with asms. Presentation at the Second Cannes ASM Workshop 1998, June 1998.

[5] A. Blass and Y. Gurevich. Abstract state machines capture parallel algorithms. Technical report, Microsoft Research, One Microsoft Way, U.S.A., November 2001. To appear in ACM ToCL.

[6] E. Börger and J. Huggins. Abstract state machines 1988 – 1998: Commented ASM bibliography. In H. Ehrig, editor, *EATCS Bulletin, Formal Specification Column*, number 64, pages 105 – 127. EATCS, February 1998.

[7] E. Börger and D. Rosenzweig. *The WAM - Definition and Compiler Correctness*, chapter 2, pages 20 – 90. Series in Computer Science and Artificial Intelligence. Elsevier Science B.V.North Holland, 1995.

[8] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editor, *Gurevich Festschrift CSL 2000*, LNCS. Springer-Verlag, 2000. to Appear.

[9] S. C. Cater and J. K. Huggins. An ASM dynamic semantics for standard ML. In Gurevich et al. [19], pages 203–223.

[10] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, Reading, MA, 1999.

[11] G. Del Castillo. Towards comprehensive tool support for abstract state machines: The asm workbench tool environment and architecture. In D. Hutter, W. Stephan, P. Treaverso, and M. Ullman, editors, *Applied Formal Methods – FM-Trends 98*, number 1641 in LNCS, pages 311–325. Springer, 1999.

[12] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[13] Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, page 317, August 1985.

[14] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Theory and Practice of Software Engineering*, pages 1–57. CS Press, 1988.

[15] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[16] Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department Technical Report, 1997.

[17] Y. Gurevich. Sequential ASM Thesis. *Bulletin of European Association for Theoretical Computer Science*, (67):93–124, February 1999. Also Microsoft Research Technical Report No. MSR-TR-99-09.

[18] Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transaction on Computational Logic*, 1(1):77–111, July 2000.

[19] Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, editors. *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[20] P. W. Kutter. Integration of the Statecharts in Specware and Aspects of Correct Oberon Code Generation. Master's thesis, ETH Zurich, 1996.

[21] P.W. Kutter. State transitions modeled as refinements. Technical report, Kestrel Institute, 1996.

[22] W. May. Specifying complex and structured systems with evolving algebras. In M. Bidoit and M. Dauchet, editors, *Proc. of TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, number 1214 in LNCS, pages 535–549. Springer, 1997.

[23] R. Milner, M. Tofte, and R. Harper. *The Definition of StandardML*. MIT Press, Cambridge, Massachusetts, 1990.

[24] M. Odersky. Programming with variable functions. In *International Conference on Functional Programming*, Baltimore, 1998. ACM.

[25] D. Pavlovic and R. Smith. Composition and refinement of behavioral specifications. In *Proceedings of 16th Automated Software Engineering Conference*, pages 157–165. IEEE press, November 2001.

[26] D. Pavlovic and R. Smith. Guarded transitions in evolving specifications. In *Proceedings of AMAST'02*, 2002.

[27] A. Tarski. Der wahrheitsbegriff in den formalisierten sprachen. *Studia Philosophica*, (1):261–405, 1936. English translation in A. Tarski. Logic, Semantics, Methamathematics. Oxford University Press.

[28] A. M. Turing. On computable numbers, with an application to the Entscheidungs problem. *Proc. London Math. Soc.*, 2(42):230–265, 1936. (Corrections on volume 2(43):544–546).

[29] J. Visser. Evolving algebras. Master's thesis, Delft University of Technology, 1996.