Formal Specification and Verification of Software

Abstract State Machines

Bernhard Beckert

Universität Koblenz-Landau
Abstract State Machines (ASMs)

**Purpose**

Formalism for modelling/formalising (sequential) algorithms

*Not:* Computability / complexity analysis

**Invented/developed by**

Yuri Gurevich, 1988

**Old name**

Evolving algebras
Features of ASMs

**Universality:** ASMs can represent all sequential algorithms
Features of ASMs

**Universality:** ASMs can represent all sequential algorithms

**Precision:** ASMs use classical mathematical structures that are well-understood
Features of ASMs

**Universality:** ASMs can represent all sequential algorithms

**Precision:** ASMs use classical mathematical structures that are well-understood

**Faithfulness:** ASMs require a minimal amount of notational coding
Features of ASMs

**Universality:** ASMs can represent all sequential algorithms

**Precision:** ASMs use classical mathematical structures that are well-understood

**Faithfulness:** ASMs require a minimal amount of notational coding

**Understandability:** ASMs use an extremely simple syntax, which can be read as a form of pseudo-code
Features of ASMs

**Universality:** ASMs can represent all sequential algorithms

**Precision:** ASMs use classical mathematical structures that are well-understood

**Faithfulness:** ASMs require a minimal amount of notational coding

**Understandability:** ASMs use an extremely simple syntax, which can be read as a form of pseudo-code

**Executability:** ASMs can be tested by executing them
Features of ASMs

**Universality:** ASMs can represent all sequential algorithms

**Precision:** ASMs use classical mathematical structures that are well-understood

**Faithfulness:** ASMs require a minimal amount of notational coding

**Understandability:** ASMs use an extremely simple syntax, which can be read as a form of pseudo-code

**Executability:** ASMs can be tested by executing them

**Scalability:** ASMs can describe a system/algorithm on different levels of abstraction
Features of ASMs

Universality: ASMs can represent all sequential algorithms

Precision: ASMs use classical mathematical structures that are well-understood

Faithfulness: ASMs require a minimal amount of notational coding

Understandability: ASMs use an extremely simple syntax, which can be read as a form of pseudo-code

Executability: ASMs can be tested by executing them

Scalability: ASMs can describe a system/algorithm on different levels of abstraction

Generality: ASMs have been shown to be useful in many different application domains
Three Postulates

Sequential Time Postulate

An algorithm can be described by defining a set of states, a subset of initial states, and a state transformation function.
Three Postulates

**Sequential Time Postulate**

An algorithm can be described by defining a set of states, a subset of initial states, and a state transformation function.

**Abstract State Postulate**

States can be described as first-order structures.
Three Postulates

**Sequential Time Postulate**

An algorithm can be described by defining a set of states, a subset of initial states, and a state transformation function.

**Abstract State Postulate**

States can be described as first-order structures.

**Bounded Exploration Postulate**

An algorithm explores only finitely many elements in a state to decide what the next state is.

There is a finite number of names (terms) for all these “interesting” elements in all states.
Example: Computing Squares

Initial State

\[ \text{square} = 0 \]
\[ \text{count} = 0 \]

**ASM for computing the square of** \textit{input}

\[
\text{if } \text{input} < 0 \text{ then}
\]
\[
\quad \text{input} := -\text{input}
\]
\[
\text{else if } \text{input} > 0 \land \text{count} < \text{input} \text{ then}
\]
\[
\quad \text{par}
\]
\[
\quad \quad \text{square} := \text{square} + \text{input}
\]
\[
\quad \quad \text{count} := \text{count} + 1
\]
\[
\text{endpar}
\]
Example: Turing Machine

\begin{verbatim}
par
    currentState := newState(currentState, content(head))
    content(head) := newSymbol(currentState, content(head))
    head := head + move(currentState, content(head))
endpar
\end{verbatim}
The Sequential Time Postulate

**Sequential algorithm**

An algorithm is associated with

- a set $S$ of states
- a set $I \subset S$ of initial states
- A function $\tau : S \rightarrow S$
  (the one-step transformation of the algorithm)
The Sequential Time Postulate

**Sequential algorithm**

An algorithm is associated with

- a set \( S \) of states
- a set \( I \subset S \) of initial states
- A function \( \tau : S \rightarrow S \) (the one-step transformation of the algorithm)

**Run (computation)**

A run (computation) is a sequence \( X_0, X_1, X_2, \ldots \) of states such that

- \( X_0 \in I \)
- \( \tau(X_i) = X_{i+1} \) for all \( i \geq 0 \)
Termination

The definition avoids the issue of termination

Possible solutions

- Add a set $F \subseteq T$ of final states
- Make the function $\tau$ partial
- Define a state $s$ to be final if $\tau(s) = s$
The Abstract State Postulate

States are first-order structures where

all states have the same vocabulary (signature)
The Abstract State Postulate

States are first-order structures where

- all states have the same vocabulary (signature)
- the transformation $\tau$ does not change the base set (universe)
The Abstract State Postulate

States are first-order structures where

- all states have the same vocabulary (signature)
- the transformation $\tau$ does not change the base set (universe)
- $S$ and $I$ are closed under isomorphism
The Abstract State Postulate

States are first-order structures where

- all states have the same vocabulary (signature)
- the transformation $\tau$ does not change the base set (universe)
- $S$ and $I$ are closed under isomorphism
- if $\zeta$ is an isomorphism from a state $X$ onto a state $Y$, then $\zeta$ is also an isomorphism from $\tau(X)$ onto $\tau(Y)$
Vocabulary (Signature)

Signatures

A signature is a finite set of function symbols, where

– each symbol is assigned an arity $n \geq 0$
– symbols can be marked *relational* (predicates)
– symbols can be marked *static* (default: dynamic)
Vocabulary (Signature)

Signatures

A signature is a finite set of function symbols, where

– each symbol is assigned an arity $n \geq 0$
– symbols can be marked *relational* (predicates)
– symbols can be marked *static* (default: dynamic)

Each signature contains

– the constant $\bot$ (“undefined”)
– the relational constants $\text{true}$, $\text{false}$
– the unary relational symbols $\text{Boole}$, $\neg$
– the binary relational symbols $\land$, $\lor$, $\rightarrow$, $\leftrightarrow$, $\equiv$

These special symbols are all static
Variables and Terms

Variables

There is an infinite set of variables

An infinite subset of these are boolean variables
Variables and Terms

Variables

There is an infinite set of variables
An infinite subset of these are boolean variables

Terms

Terms are build as usual from variables and function symbols
A term is boolean if
– it is a boolean variable or
– its top-level symbol is relational
First-order Structures (States)

First-order structures (states) consist of

- a non-empty universe (called $\text{BaseSet}$)

- an interpretation $I$ of the symbols in the signature
First-order Structures (States)

First-order structures (states) consist of

- a non-empty universe (called BaseSet)
- an interpretation $I$ of the symbols in the signature

Restrictions on states

- $tt, ff, \bot \in \text{BaseSet}$ (different elements)
- $I(\text{true}) = tt$
- $I(\text{false}) = ff$
- $I(\bot) = \bot$
- If $f$ is relational, then $I(f) : \text{BaseSet} \rightarrow \{tt, ff\}$
- $I(\text{Boole}) = \{tt, ff\}$
- $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, =$ are interpreted as usual
The Reserve of a State

**Reserve**

Consists of the elements that are “unknown” in a state
The Reserve of a State

Reserve

Consists of the elements that are “unknown” in a state

An element $a$ is in the reserve if:

- If $f$ is relational, then $I(f)(a) = ff$
- If $f$ is not relational, then $I(f)(a) = \bot$
- For no function symbol $f$ is $a$ in the domain of $I(f)$
The Reserve of a State

**Reserve**

Consists of the elements that are “unknown” in a state

An element $a$ is in the reserve if:

1. If $f$ is relational, then $I(f)(a) = ff$
2. If $f$ is not relational, then $I(f)(a) = \bot$
3. For no function symbol $f$ is $a$ in the domain of $I(f)$

**Definition**

The reserve of a state must be infinite
Extended States

Variable assignment

A function

\[ \beta : \text{Var} \rightarrow \text{BaseSet} \]

(boolean variables are assigned \( \text{tt} \) or \( \text{ff} \))

Extended state

A pair

\[ (X, \beta) \]

consisting of a state \( X \) and a variable assignment \( \beta \)
Evaluation of Terms

Given: Extended state \((X, \beta)\)

Evaluation of terms

The evaluation of terms in an extended states is defined by:

\[
(X, \beta)(x) = \beta(x) \quad \text{for variables } x
\]

\[
(X, \beta)f(s_1, \ldots, s_n) = I(f)((X, \beta)(s_1), \ldots, (X, \beta)(s_n))
\]

where \(I\) is the interpretation function of \(X\)
Evaluation of Terms

Given: Extended state \((X, \beta)\)

**Evaluation of terms**

The evaluation of terms in an extended states is defined by:

\[
(X, \beta)(x) = \beta(x) \quad \text{for variables } x
\]

\[
(X, \beta)f(s_1, \ldots, s_n) = I(f)((X, \beta)(s_1), \ldots, (X, \beta)(s_n))
\]

where \(I\) is the interpretation function of \(X\)

**Notation**

\(f^X\) for \(I(f)\)

\(t^X\) for \((X, \beta)(t)\) if \(t\) is a ground term
Example: Trees

**Vocabulary**

- **nodes**: unary, boolean: the class of nodes (type/universe)
- **strings**: unary, boolean: the class of strings
- **parent**: unary: the parent node
- **firstChild**: unary: the first child node
- **nextSibling**: unary: the first sibling
- **label**: unary: node label
- **c**: constant: the current node
**Example: Trees**

**Terms**

\[
\text{parent}(\text{parent}(c)) \\
\text{label}((\text{firstChild}(c)) \\
\text{parent}(\text{firstChild}(c)) = c \\
\text{nodes}(x) \rightarrow \text{parent}(x) = \text{parent}(\text{nextSibling}(x))
\]

(*x is a variable*)
Isomorphism of States

Isomorphism

A bijection $\zeta$ from $X$ to $Y$ is an isomorphism if:

- for all symbols $f$
- all $a_1, \ldots, a_n \in \text{BaseSet}(X)$

$$\zeta(f^X(a_1, \ldots, a_n)) = f^Y(\zeta(a_1), \ldots, \zeta(a_n))$$
Isomorphism of States

**Isomorphism**

A bijection $\zeta$ from $X$ to $Y$ is an isomorphism if:

- for all symbols $f$
- all $a_1, \ldots, a_n \in \text{BaseSet}(X)$

\[
\zeta(f^X(a_1, \ldots, a_n)) = f^Y(\zeta(a_1), \ldots, \zeta(a_n))
\]

**Equivalent condition:**

\[
f^X(a_1, \ldots, a_n) = b \quad \text{iff} \quad f^Y(\zeta(a_1), \ldots, \zeta(a_n)) = \zeta(b)
\]
Isomorphism of States

**Lemma (Isomorphism)**

Isomorphic states are indistinguishable by ground terms:

\[ \zeta(t^X) = t^Y \quad \text{for all ground terms} \; t \]

\[ (t = s)^X = tt \iff (t = s)^Y = tt \quad \text{for all ground terms} \; s, t \]
Isomorphism of States

Lemma (Isomorphism)

Isomorphic states are indistinguishable by ground terms:

\[ \zeta(t^X) = t^Y \quad \text{for all ground terms } t \]

\[ (t = s)^X = tt \iff (t = s)^Y = tt \quad \text{for all ground terms } s, t \]

Justification for postulate

If \( \zeta \) is an isomorphism from a state \( X \) onto a state \( Y \),
then \( \zeta \) is also an isomorphism from \( \tau(X) \) onto \( \tau(Y) \)

Algorithm must have the same behaviour for indistinguishable states

Isomorphic states are different representations of the same abstract state!
Isomorphism of States: Example

Vocabulary

constants (dynamic): \( a, b, \text{count} \)
unary functions (dynamic): \( f, g \)
static functions: \( 1, + \)

Algorithm

\[
\text{par} \\
\quad \text{if } a = b \text{ then } \text{count} := \text{count} + 1 \\
\quad \text{else skip} \\
\text{endif} \\
\quad a := f(a) \\
\quad b := g(b) \\
\text{endpar}
\]

Initial State

\( \text{count} = 0 \)
State Updates

Locations

A location is a pair

\[(f, \vec{a})\]

with

- \(f\) an \(n\)-ary function symbol
- \(\vec{a} \subseteq \text{BaseSet}\) an \(n\)-tuple
State Updates

Locations

A location is a pair

\[(f, \vec{a})\]

with

- \(f\) an \(n\)-ary function symbol
- \(\vec{a} \subset \text{BaseSet}\) an \(n\)-tuple

Examples

\[(\text{parent}, \langle a \rangle), (\text{firstChild}, \langle a \rangle), (\text{nextSibling}, \langle a \rangle), (c, \langle \rangle)\]

are locations \((a\) is an element from \(\text{BaseSet}_{\text{Tree}}\))
State Updates

Updates

An update is a triple

\[(f, \vec{a}, b)\]

with

- \((f, \vec{a})\) a location
- \(f\) not static
- \(b \in \text{BaseSet}\)
- if \(f\) is relational, then \(b \in \{tt, ff\}\)
State Updates

Updates

An update is a triple

\[(f, \vec{a}, b)\]

with

- \((f, \vec{a})\) a location
- \(f\) not static
- \(b \in \text{BaseSet}\)
- if \(f\) is relational, then \(b \in \{tt, ff\}\)

Trivial update

An update is trivial if \(f^X(\vec{a}) = b\)
State Updates: Consistency

**Clash**

Two updates

\[(f_1, \vec{a}_1, b_1) \quad (f_2, \vec{a}_2, b_2)\]

clash if

\[(f_1, \vec{a}_1) = (f_2, \vec{a}_2) \quad \text{but} \quad b_1 \neq b_2\]
State Updates: Consistency

Clash

Two updates

\((f_1, \vec{a}_1, b_1)\) \quad \((f_2, \vec{a}_2, b_2)\)

clash if

\((f_1, \vec{a}_1) = (f_2, \vec{a}_2)\) \quad \text{but} \quad b_1 \neq b_2

Example

These two updates clash: \((\text{nodes}, a, tt)\) \quad \((\text{nodes}, a, ff)\)
State Updates: Consistency

Clash

Two updates

\[(f_1, \vec{a}_1, b_1) \quad (f_2, \vec{a}_2, b_2)\]

clash if

\[(f_1, \vec{a}_1) = (f_2, \vec{a}_2) \quad \text{but} \quad b_1 \neq b_2\]

Example

These two updates clash:

\[(nodes, a, tt) \quad (nodes, a, ff)\]

Consistent set of updates

A set of updates is consistent if it does not contain clashing updates
State Updates: Execution

Executing an update

An update is executed by changing the value of $f^X(\vec{a})$ to $b$
State Updates: Execution

**Executing an update**

An update is executed by changing the value of $f^X(\vec{a})$ to $b$

**Executing a set of updates**

A consistent set of updates is executed by **simultaneously** executing all updates in the set

An inconsistent set of updates is executed by doing nothing
State Updates: Execution

Executing an update

An update is executed by changing the value of $f_X^X(\vec{a})$ to $b$

Executing a set of updates

A consistent set of updates is executed by simultaneously executing all updates in the set

An inconsistent set of updates is executed by doing nothing

Notation

The result of executing a set $\Delta$ of updates in a state $X$ is denoted with $X + \Delta$
State Updates: Uniqueness

Lemma (State Update Uniqueness)

$X, Y$ states with
- the same vocabulary
- the same base set

Then there is exactly one consistent set $\Delta$ of non-trivial updates such that

$$Y = X + \Delta$$
Lemma (State Update Uniqueness)

$X$, $Y$ states with

- the same vocabulary
- the same base set

Then there is exactly one consistent set $\Delta$ of non-trivial updates such that

$$Y = X + \Delta$$

Notation

We write $\Delta(X)$ for the set of updates such that

$$\tau(X) = X + \Delta(X)$$
The Bounded Exploration Postulate

There is a finite set $T$ of ground terms for such that for all states $X, Y$:

If

$$t^X = t^Y \text{ for all } t \in T$$

then

$$\Delta(X) = \Delta(Y)$$
The Bounded Exploration Postulate

There is a finite set $T$ of ground terms for such that for all states $X, Y$: If

$$t^X = t^Y \text{ for all } t \in T$$

then

$$\Delta(X) = \Delta(Y)$$

**Bounded exploration witness**

If such a set $T$ is closed under the sub-term relation, it is called a bounded exploration witness
Bounded Exploration: Example

Algorithm given by

\[
\text{if } p(c) \text{ then } c := s(c)
\]

Bounded exploration witness

\[
\{ c, s(c), p(c) \}
\]
Bounded Exploration: Counter Examples

“Algorithms” *not* satisfying the bounded exploration postulate

\[
\text{for all } x, y \text{ with } edge(x, y) \land \text{reachable}(x) \land \neg\text{reachable}(y) \\text{ do}
  \begin{align*}
    \text{reachable}(y) &:= \text{true} \\
  \end{align*}
\text{enddo}
\]
“Algorithms” *not* satisfying the bounded exploration postulate

\[
\text{for all} \ x, y \ \text{with} \ edge(x, y) \land \text{reachable}(x) \land \neg \text{reachable}(y) \\
\text{do} \\
\text{reachable}(y) := \text{true} \\
\text{enddo}
\]

**Bounded change is not enough**

\[
\text{if} \ \forall x \exists y \ edge(x, y) \ \text{then} \\
\text{hasIsolatedPoints} := \text{false} \\
\text{else} \\
\text{hasIsolatedPoints} := \text{true} \\
\text{endif}
\]
Lemma (Accessibility Lemma)

Given a bounded exploration witness $T$

If

$$(f, \langle a_1, \ldots, a_n \rangle, a_0) \in \Delta(X)$$

then there are terms $t_0, \ldots, t_n \in T$ such that

$$t_i^X = a_i \quad \text{for } 0 \leq i \leq n$$
Accessibility Lemma

Lemma (Accessibility Lemma)

Given a bounded exploration witness $T$

If

$$(f, \langle a_1, \ldots, a_n \rangle, a_0) \in \Delta(X)$$

then there are terms $t_0, \ldots, t_n \in T$ such that

$$t_i^X = a_i \quad \text{for } 0 \leq i \leq n$$

Corollary

There is a finite limit on the size of $\Delta(X)$, which does not depend on $X$
Update Rules

An update rule has the form

\[ f(s_1, \ldots, s_n) := t \]

where

– \( f \) is a function symbol of arity \( n \)
– \( s_1, \ldots, s_n, t \) and \( t \) are ground terms
Update Rules

An update rule has the form

\[ f(s_1, \ldots, s_n) := t \]

where

- \( f \) is a function symbol of arity \( n \)
- \( s_1, \ldots, s_n, t \) and \( t \) are ground terms

Executing an update rule

An update rule \( R \) is executed in state \( X \) by executing the update set

\[ R(X) = \{ (f, \langle s_1^X, \ldots, s_n^X \rangle, t^X) \} \]
Note

The interpretation $g^X$ of function symbols $g$ occurring in an update rule

$$f(s_1, \ldots, s_n) := t$$

in the $s_i$ or in $t$ can be

- an “external” static function defined in the initial state
- of high computational complexity
- even non-computable

This allows to describe algorithms on arbitrary levels of abstraction
A block rule has the form

\[
\begin{align*}
\text{par} & \\
R_1 & \\
R_2 & \\
\vdots & \\
R_k & \\
\text{endpar}
\end{align*}
\]

\textbf{where } R_1, \ldots, R_k \textbf{ are rules } (k \geq 0)
A block rule has the form

\[
\text{par} \quad R_1 \quad R_2 \quad \ldots \quad R_k \quad \text{endpar}
\]

where \( R_1, \ldots, R_k \) are rules \((k \geq 0)\)

**Executing a block rule**

A block rule \( R \) is executed in state \( X \) by executing the update set

\[
R(X) = R_1(X) \cup \ldots \cup R_k(X)
\]
The empty block is written as

```
skip
```
State Update Representation Lemma

Consequence of the Accessibility Lemma

**Lemma (State Update Representation)**

For every state $X$, there is a block rule $R_X$ such that

$$R_X(X) = \Delta(X)$$
Consequence of the Accessibility Lemma

**Lemma (State Update Representation)**

For every state $X$, there is a block rule $R_X$ such that

$$R_X(X) = \Delta(X)$$

**Note**

In general

$$R_X(Y) \neq \Delta(Y)$$
$T$-Similar States

$T$-similarity

Given a bounded exploration witness $T$

States $X, Y$ are $T$-similar if for all $t_1, t_2 \in T$:

$$t^X_1 = t^X_2 \quad \text{iff} \quad t^Y_1 = t^Y_2$$
$T$-Similar States

$T$-similarity

Given a bounded exploration witness $T$

States $X, Y$ are $T$-similar if for all $t_1, t_2 \in T$:

$$t_1^X = t_2^X \quad \text{iff} \quad t_1^Y = t_2^Y$$

Note

$T$-similar states $X, Y$ are “isomorphic” on $T^X$ resp. $T^Y$
$T$-Similar States

$T$-similarity

Given a bounded exploration witness $T$

States $X, Y$ are $T$-similar if for all $t_1, t_2 \in T$:

$$t_1^X = t_2^X \text{ iff } t_1^Y = t_2^Y$$

Note

$T$-similar states $X, Y$ are “isomorphic” on $T^X$ resp. $T^Y$

Lemma ($T$-similarity)

There is a finite number of states $X_1, \ldots, X_m$ such that every state is $T$-similar to one of the $X_i$
Lemma \((T\text{-similarity Representation})\)

There is a relational term \(\phi_X\) such that

\(\phi_X\) is true in \(Y\) iff \(Y\) is \(T\)-similar to \(X\)
**Conditional State Update Representation Lemma**

**Lemma ($T$-similarity Representation)**
There is a relational term $\phi_X$ such that
$\phi_X$ is true in $Y$ iff $Y$ is $T$-similar to $X$

**Lemma (Conditional State Update Representation)**
If $X, Y$ are $T$-similar, then

$$R_X(Y) = \Delta(Y)$$
An if rule has the form

\[
\text{if } cnd \text{ then } R_1 \\
\text{else } R_2 \\
\text{endif}
\]

where \( R_1, R_2 \) are rules and \( cnd \) is a relational term.
If Rule

An if rule has the form

\[
\text{if } \textit{cnd} \text{ then } R_1 \text{ else } R_2 \text{ endif}
\]

where \( R_1, R_2 \) are rules and \( \textit{cnd} \) is a relational term

Executing an if rule

An if rule \( R \) is executed in state \( X \) by executing the update set

\[
R(X) = \begin{cases} 
R_1(X) & \text{if } \textit{cond}^X = \text{tt} \\
R_2(X) & \text{otherwise}
\end{cases}
\]
Main Theorem

**Theorem**

For every algorithm there is a rule $\mathcal{R}$ such that

$$\mathcal{R}(X) = \Delta(X) \text{  for all states } X$$
Main Theorem

Theorem

For every algorithm there is a rule $R$ such that

$$R(X) = \Delta(X) \quad \text{for all states } X$$

Proof

An example for such a rule is

if $\phi_{X_1}$ then $R_{X_1}$
else if $\phi_{X_2}$ then $R_{X_2}$
::
else if $\phi_{X_m}$ then $R_{X_m}$
endif . . . endif
An abstract state machine representing an algorithm consists of

- the rule (program) $R$ such that

$$R(X) = \Delta(X) \quad \text{for all states } X$$

- the set of states of the algorithm

- the set of initial states of the algorithm
Abstract State Machine Representing an Algorithm

An abstract state machine representing an algorithm consists of

- the rule (program) \( R \) such that

\[
R(X) = \Delta(X) \quad \text{for all states } X
\]

- the set of states of the algorithm
- the set of initial states of the algorithm

Note

The interpretation of static functions is “built into” the initial states
ASM Applications

- Abstract Algorithms
  Lamport’s Bakery Algorithm

- Architectures
  Pipelining in the ARM2 RISC Microprocessor
  Hennessey and Patterson DLX pipelined microprocessor

- Benchmark Examples
  Production Cell Control Problem
  Steam Boiler Problem

- Compiler Correctness
  Compiling Occam to Transputer code

- Databases
  Formalization of Database Recovery
ASM Applications

- Distributed Systems
  Communicating evolving algebras

- Hardware
  Specification of the DEC-Alpha Processor Family

- Java
  Semantics of Java
  Defining the Java Virtual Machine
  Investigating Java Concurrency

- Logic & Computability
  Linear Time Hierarchy Theorems for ASMs

- Mechanical Verification
  Model Checking Support for the ASM
  Mechanical verification of the correctness proof in WAM Case Study
ASM Applications

(Other) Models of Computation
Investigating the formal relation between
– ASMs and Predicate Transition Nets
– ASM and Schönhage Storage Modification Machines

Montages
A version of ASMs for specifying static and dynamic semantics of programming languages
Combines graphical and textual elements to yield specifications similar in structure, length, and complexity to those in common language manuals

Natural Languages
Mathematical Models of Language
ASM Applications

- **Programming Languages**
  Operational semantics of
  Prolog, Parlog, C, C++, COBOL, Occam, Oberon

- **Real-time Systems**
  Railway crossing system

- **Security**
  Formal analysis of the Kerberos Authentication System

- **VHDL**
  Semantical analysis of VHDL-AMS
Features of ASMs Revisited

**Universality:** ASMs can be represent all sequential algorithms

**Precision:** ASMs use classical mathematical structures that are well-understood

**Faithfulness:** ASMs require a minimal amount of notational coding

**Understandability:** ASMs use an extremely simple syntax, which can be read as a form of pseudo-code

**Executability:** ASMs can be tested by executing them

**Scalability:** ASMs can describe a system/algorithms on different levels of abstraction

**Generality:** ASMs have been shown to be useful in many different application domains