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

Executablity: 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

Executablity: 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

Executablity: 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 algortihm 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 algortihm 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 algortihm 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

$square = 0$
$count = 0$

ASM for computing the square of $input$

if $input < 0$ **then**
  $input := -input$
else if $input > 0 \wedge count < input$ **then**
 par
  $square := square + input$
  $count := count + 1$
 endpar

Example: Turing Machine

par

$$\begin{array}{lcl} currentState & := & newState(currentState, content(head)) \\ content(head) & := & newSymbol(currentState, content(head)) \\ head & := & head + move(currentState, content(head)) \end{array}$$

endpar

The Sequential Time Postulate

Sequential algorithm

An algorithm is associated with

- **a set $\mathcal{S}$ of states**
- **a set $I \subset \mathcal{S}$ of initial states**
- **A function** $\tau : \mathcal{S} \longrightarrow \mathcal{S}$
  (the one-step transformation of the algorithm)

The Sequential Time Postulate

Sequential algorithm

An algorithm is associated with

- **a set $\mathcal{S}$ of states**
- **a set $I \subset \mathcal{S}$ of initial states**
- **A function** $\tau : \mathcal{S} \longrightarrow \mathcal{S}$
  (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 $\mathcal{F} \subset \mathcal{T}$ of final states**
- **Make the function τ 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 τ 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 τ does not change the base set (universe)**
- **$\mathcal{S}$ and $\mathcal{I}$ are closed under isomorphism**

The Abstract State Postulate

States are first-order structures where

- all states have the same vocabulary (signature)
- the transformation τ does not change the base set (universe)
- $\mathcal{S}$ and $\mathcal{I}$ are closed under isomorphism
- if ζ is an isomorphism from a state X onto a state Y,
  then ζ 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 $\perp$ ("undefined")**
- **the relational constants true, false**
- **the unary relational symbols $Boole$, $\neg$**
- **the binary relational symbols $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $=$**

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 $\mathrm{BaseSet}$)**
- **an interpretation I of the symbols in the signature**

First-order Structures (States)

First-order structures (states) **consist of**

- **a non-empty universe (called $\mathrm{BaseSet}$)**
- **an interpretation I of the symbols in the signature**

Restrictions on states

- $tt, \mathit{ff}, \perp \in \mathrm{BaseSet}$ **(different elements)**
- $I(\mathbf{true}) = tt$
- $I(\mathbf{false}) = \mathit{ff}$
- $I(\perp) = \perp$
- **If f is relational, then** $I(f) : \mathrm{BaseSet} \rightarrow \{tt, \mathit{ff}\}$
- $I(\mathrm{Boole}) = \{tt, \mathit{ff}\}$
- $\neg$**,** $\wedge$**,** $\vee$**,** $\rightarrow$**,** $\leftrightarrow$**,** $=$ **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 $I(f)(a) = \mathit{ff}$**
- **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:

- If f is relational, then $I(f)(a) = \mathit{ff}$
- If f is not relational, then $I(f)(a) = \perp$
- 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

$$\beta : \mathit{Var} \rightarrow \mathrm{BaseSet}$$

(boolean variables are assigned tt or ff)

Extended state

A pair

$$(X, \beta)$$

consisting of a state X and a variable assignment β

Evaluation of Terms

Given: **Extended state** (X, β)

Evaluation of terms

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

- $(X, \beta)(x) = \beta(x)$ **for variables** x
- $(X, \beta)f(s_1, ..., s_n) = I(f)((X, \beta)(s_1), \ldots, (X, \beta)(s_n))$

where I **is the interpretation function of** X

Evaluation of Terms

Given: **Extended state** (X, β)

Evaluation of terms

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

- $(X, \beta)(x) = \beta(x)$ **for variables** x
- $(X, \beta)f(s_1, ..., s_n) = I(f)((X, \beta)(s_1), \ldots, (X, \beta)(s_n))$

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

$parent(parent(c))$

$label(firstChild(c))$

$parent(firstChild(c)) = c$

$nodes(x) \to parent(x) = parent(nextSibling(x))$

(x is a variable)

Isomorphism of States

Isomorphism

A bijection ζ 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 ζ from X to Y is an isomorphism if:

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

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

Equivalent condition:

$$f^X(a_1, \ldots, a_n) = b \quad \textbf{iff} \quad f^Y(\zeta(a_1), \ldots, \zeta(a_n)) = \zeta(b)$$

Isomorphism of States

Lemma (Isomorphism)

Isomorphic states are indistinguishable by ground terms:

- $\zeta(t^X) = t^Y$ **for all ground terms** t
- $(t = s)^X = tt$ **iff** $(t = s)^Y = tt$ **for all ground terms** s, t

Isomorphism of States

Lemma (Isomorphism)

Isomorphic states are indistinguishable by ground terms:

- $\zeta(t^X) = t^Y$ **for all ground terms** t
- $(t = s)^X = tt$ **iff** $(t = s)^Y = tt$ **for all ground terms** s, t

Justification for postulate

If ζ is an isomorphism from a state X onto a state Y,
then ζ 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): $a, b, count$

unary functions (dynamic): f, g

static functions: $1, +$

Algorithm

par

  if $a = b$ **then** $count := count + 1$

    else skip

  endif

  $a := f(a)$

  $b := g(b)$

endpar

Initial State

$count = 0$

State Updates

Locations

A location is a pair

$$(f, \vec{a})$$

with

- f **an** n**-ary function symbol**
- $\vec{a} \subset \mathrm{BaseSet}$ **an** n**-tuple**

State Updates

Locations

A location is a pair

$$(f, \vec{a})$$

with

- **–** f **an** n**-ary function symbol**
- **–** $\vec{a} \subset \mathrm{BaseSet}$ **an** n**-tuple**

Examples

$$(parent, \langle a \rangle), \quad (firstChild, \langle a \rangle), \quad (nextSibling, \langle a \rangle), \quad (c, \langle\rangle)$$

are locations **(**a **is an element from** $\mathrm{BaseSet}_{\mathrm{Tree}}$**)**

State Updates

Updates

An update is a triple

$$(f, \vec{a}, b)$$

with

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

State Updates

Updates

An update is a triple

$$(f, \vec{a}, b)$$

with

- $(f, \vec{a})$ **a location**
- f **not static**
- $b \in \mathrm{BaseSet}$
- **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

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

clash if

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

State Updates: Consistency

Clash

Two updates

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

clash if

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

Example

These two updates clash: $(nodes, a, tt) \quad (nodes, a, ff)$

State Updates: Consistency

Clash

Two updates

$$(f_1, \vec{a_1}, b_1) \qquad (f_2, \vec{a_2}, b_2)$$

clash if

$$(f_1, \vec{a_1}) = (f_2, \vec{a_2}) \quad \textbf{but} \quad b_1 \neq b_2$$

Example

These two updates clash: $(nodes, a, tt) \quad (nodes, a, f\!f)$

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 $f^X(\vec{a})$ 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 Δ 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 Δ of non-trivial updates such that

$$Y = X + \Delta$$

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 Δ 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

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

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

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

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

$$\textbf{if } p(c) \textbf{ then } c := s(c)$$

Bounded exploration witness

$$\{ \; c, \; s(c), \; p(c) \; \}$$

Bounded Exploration: Counter Examples

"Algorithms" *not* satisfying the bounded exploration postulate

for all x, y **with** $edge(x, y) \land reachable(x) \land \neg reachable(y)$
do
$reachable(y) := \mathbf{true}$
enddo

Bounded Exploration: Counter Examples

"Algorithms" *not* satisfying the bounded exploration postulate

for all x, y **with** $edge(x, y) \land reachable(x) \land \neg reachable(y)$
  do
    $reachable(y) := \mathbf{true}$
  enddo

Bounded change is not enough

if $\forall x \exists y \, edge(x, y)$ **then**
  $hasIsolatedPoints := \mathbf{false}$
else
  $hasIsolatedPoints := \mathbf{true}$
endif

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**

$$t_i^X = a_i \qquad \textbf{for } 0 \leq i \leq n$$

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

$$t_i^X = a_i \qquad \textbf{for } 0 \leq i \leq n$$

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**

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

Update Rules: Computability and Complexity

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

Block Rules

A block rule has the form

> **par**
> R_1
> R_2
> $\vdots$
> R_k
> **endpar**

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

Block Rules

A block rule has the form

par
R_1
R_2
$\vdots$
R_k
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

$$R(X) \;=\; R_1(X) \cup \ldots \cup R_k(X)$$

Empty Block

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

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

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

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

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 $t_1, t_2 \in T$:

$$t_1^X = t_2^X \qquad \textbf{iff} \qquad t_1^Y = t_2^Y$$

T-Similar States

T-similarity

Given a bounded exploration witness T

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

$$t_1^X = t_2^X \qquad \textbf{iff} \qquad t_1^Y = t_2^Y$$

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 $t_1, t_2 \in T$:

$$t_1^X = t_2^X \qquad \textbf{iff} \qquad t_1^Y = t_2^Y$$

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

Conditional State Update Representation Lemma

Lemma (T-similarity Representation)

There is a relational term ϕ_X such that

ϕ_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 ϕ_X such that

ϕ_X is true in Y iff Y is T-similar to X

Lemma (Conditional State Update Representation)

If X, Y are T-similar, then

$$R_X(Y) = \Delta(Y)$$

If Rule

An if rule has the form

if	cnd	**then**	R_1
		else	R_2
endif			

where R_1, R_2 **are rules and** cnd **is a relational term**

If Rule

An if rule has the form

if	cnd	**then**	R_1
		else	R_2
endif			

where R_1, R_2 **are rules and** cnd **is a relational term**

Executing an if rule

An if rule R **is executed in state** X **by executing the update set**

$$R(X) = \begin{cases} R_1(X) & \textbf{if } cond^X = tt \\ R_2(X) & \textbf{otherwise} \end{cases}$$

Main Theorem

Theorem

For every algorithm there is a rule R such that

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

Main Theorem

Theorem

For every algorithm there is a rule R such that

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

Proof

An example for such a rule is

$$\begin{array}{llll} \textbf{if} & \phi_{X_1} & \textbf{then} & R_{X_1} \\ \textbf{else if} & \phi_{X_2} & \textbf{then} & R_{X_2} \\ \vdots & & & \\ \textbf{else if} & \phi_{X_m} & \textbf{then} & R_{X_m} \\ \multicolumn{4}{l}{\textbf{endif} \ldots \textbf{endif}} \end{array}$$

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

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

Executablity: 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