Development of Rigorous Adaptive Information Systems

Dr. Nasreddine Aoumeur
FIN, ITI, DB group
aoumeur@iti.cs.uni-magdeburg.de

Course Site:
wwwiti.cs.uni-magdeburg.de/~aoumeur
wwwiti.cs.uni-magdeburg.de/iti_db/lehre/oois/inde
Information Systems: Working definition

- *reactive* systems (i.e. in continuous interaction with their environment), with
- large amount of *immutable* and *non-immutable data* (i.e. fixed and changing) and, with
- *processes and activities* for exhibiting behaviors on these (state-less and -full) data.
Conceptual Modelling of Information Systems

IS Conceptual Model

State-less and -ful DATA

Processes and Rules

- Entity-Relationship
- Object-oriented paradigm
- UML class-diagrams
- UML use-cases

- Petri Nets
- UML Sequence diagrams
- UML state-charts diagrams
Data types problems in E/R and OO and UML

E/R Conceptual Model

Customer
- Name: String
- Birth-Date: Date
- Address: Address
- Income: Real+
- OpenDt: Date

(Running) Account
- Number: Nat
- Balance: Real+
- Limit: Real+
- History: List[Date, Real+]

Programming (java, C++)

String: Any concatenation of letters/symbols/numbers
Date: Internal representation of date
Real+: [-Minreal, +Maxreal] = [-e-42, +e+42]

All these datatypes are implementation-based
Specification of Abstract data-types (ADT)

- **Data** serves for representing **Information**.
  - It forms the basis for information processing.
- Data are usually described using
  - symbols, for building texts, tables, graphics, etc.
- For elementary information like numbers, letters and very useful information like phonebooks, train-timetables, etc, there have been some representation convention.
- But generally, they is **hardly unified conventions, norms or rules** for representing information.
• **The internal machine representation does not follow usually the conventional representation:**

• **Example:**
  - numbers are in decimal but in machine are rather binary.
  - Phonebooks are organized using the names, whereas in the machine are based more on hashtables or search in trees.

• **The internal representation is directed by the efficiency in time and space, whereas the external representation SHOULD be based more on ergonomy and users suitability.**
ADT: a general introduction

• How appropriate is a given data representation depends mainly on **where and how** such data will be later used.

• That is, **which operations** will be performed on them. This of course is to become **more and precise along the life-cycle development**.

• So it is **very important to delay any final representation as much as possible, until all software aspects are clearly determined**.

In other words, **no specific internal representation at the specification level for instance**.
ADT: a general introduction

- During the first phases of software development cycle we have to:
  - describe the data in “abstract” way, i.e. independent of any representation.
  - In this case different concrete representation of the same data can be derived later depending on the need of managers, users, designers, coders as well as on the specificity of the used machines.

- This also means that for data description, a spectrum of expression mechanisms have to be available in each development phase.
Data types: a general introduction

- For describing information one need a large amount of data elements,
  - like numbers, letters, points, lines, etc.
- Usually the whole intended elements are gathered in Types, where each type determines the set of belonging elements.
- The criterion for data type is that all its data elements have the same properties.
- In programming languages this allows fixing the arguments of functions or procedures.
- In software modelling, the purpose of typing is generally for using such data types for operations.
Data types : a general introduction

• So we can say that around some data types different sets of operations may be declared.

• Example: With a set of persons (as data type), we may have different operations:
  – sorting on names, sorting on ages, sorting on the numbers of children, sorting on addresses, etc.

• From this perception, we can understand that operations are an intrinsic part of any data type.

a data type is rather a set of values with a given set of operations.
Data types : a general introduction

• Example 1: A data type BOOL1 consists of a set of values: \texttt{bool} = \{true, false\} and the boolean operations \(\neg\) and \(\lor\).
  
  » \(\neg: \texttt{bool} \rightarrow \texttt{bool}\),
  
  » \(\lor: \texttt{bool} \times \texttt{bool} \rightarrow \texttt{bool}\)
  
  » There are also boolean operations :\(\land, \Rightarrow, \Leftrightarrow, ...\) With different parameters.

• Generally, an \(n\)-ary operation \(f\) with a set of values \(W\) is represented as: \(f: W^n \rightarrow W\). This is applicable also for \(0\)-ary functions called constants.
Data types: a general introduction

Example 2: Let a data type BOOL2 consisting of set of the boolean operations $\neg$ and $\lor$.

- $\text{true} : \rightarrow \text{bool}$,
- $\text{false} : \rightarrow \text{bool}$,
- $\neg : \text{bool} \rightarrow \text{bool}$,
- $\lor : \text{bool} \times \text{bool} \rightarrow \text{bool}$

There are also boolean operations $\land, \Rightarrow, \equiv, ...$ With different parameters.

Example 3: Let NAT be the natural numbers composed of the usual set of elements: $\text{nat} = \{0, 1, 2, ...\}$ with the operations:

- $0 : \rightarrow \text{nat}$,
- $\text{succ} : \text{nat} \rightarrow \text{nat}$

The operation $\text{succ}$ delivers the successor, i.e. $\text{succ}(n)=n+1$. Addition (+), multiplication (x), etc ... can also be in nat.
Data types : a general introduction

• Example 4 : To these natural operations, we can add for instance the comparison operation $\leq$.

  » $\leq : \text{nat} \times \text{nat} \rightarrow \text{bool}$,
  
  – Here, we get a heterogeneous Data type $(\text{BOOL1} + \text{NAT})$. Very frequent.

• Example 5 : For defining an array, we can have

  » $A$ : for the data types of arrays,
  » $E$ : the data type of the array elements,
  » $I$ : as a data type for the indices of $A$.

  – With $W_A$, $W_E$, $W_I$ their respective set of elements, we may have the assignment operation as:

  » assign : $W_A \times W_E \times W_I \rightarrow W_A$

  – So for instance, assign($a$, $i$, $e$), means $a[i] := e$. 
Data types: a general introduction

- For coping with realistic problems, we rather speak about a family of inter-related data types incrementally conceived.
- Such inter-related data types $D_1, \ldots, D_n$ are to be composed of:
  - A set of values $W_1, \ldots, W_n$
  - A set of operations having their arguments and results in $W_1, \ldots, W_n$
- Such structures are known in mathematic as algebra. $W_1, \ldots, W_n$ are called the domains of the algebra and the operations are the associations or operations on them.
- Group, Monoid, etc are some homogeneous algebras; vectors, matrix are heterogeneous ones.
For our purpose **Algebras** represent the adequate precise mathematical framework.

That is, each **family of data types** will be regarded as an algebra with appropriate domains and operations.
Abstract Data types: an introduction

• While specifying application-oriented data types, in most cases we could not or we are not interested on a complete description.

• We focus generally only on the description of main properties that should be always present.

• This provides more choices for later steps: either we can add more properties or refine existing ones.

• Under abstract data types we understand such incomplete, with free choice, characterization of the main properties of a given data type.
Abstract Data types : an introduction

• Example 6 : For conceiving a sorting process, we require just to specify that
  – an order or an arrangement is declared on the elements to be ordered.
• Whether such elements are naturals, integers, rationals, reals
  – with their natural-based sorting,
  – or even strings with their lexicographical-based sorting to be sorted is not really premordiale at an abstract level.
• The required abstract data types here as an ordered set of finite elements (SORT-ELT).
Abstract Data types : an introduction

• Example 7 : to conceive a counter modulo 16, we require the characterization of an abstract data type \textbf{COUNT} with the properties:
  
  – There should be two operations:
    
    » \textbf{reset} : \quad \rightarrow \quad \text{count}
    
    » \textbf{Increment} : \quad \rightarrow \quad \text{count}
  
  – 16 applications of \textbf{increment} after a \textbf{reset} have to bring the counter to the same state of reset. All inbetween values should be distinct.

  – Any value in counter has to be reached using reset and increment.

• So we can have, for instance, as counter-based data types: $0,\ldots,15$ and \textbf{increment}(n)=\text{n+1} for \text{n in \{0,\ldots,15\}}. But we can also have $0,-1,\ldots,-15$ and \textbf{increment}(n)=\text{n-1}; or $92, 94,\ldots,120$ and \textbf{increment}(n) = \text{n+2}. They all are \textbf{isomorph} under renaming.
Abstract Data types: an introduction

An abstract data type is a class of data types (or a category of algebras).
• The signature of an algebraic specification consists of:
  – sort symbols of some data and
  – operation symbols on them

• **Definition 1 (Signature)**: A signature is a pair $\Sigma = \langle S, F \rangle$. With $S$ a set of sorts, and $F = \{f_{s_1, \ldots, s_n, s}\}$, with $s_i \in F$ is a set of $S^xS$-indexed symbol of operations. Each operation is usually represented as $f : s_1 \times \ldots \times s_n \rightarrow s$
Algebraic specifications: formal definitions

• Signature may be graphically represented as graphs with:
  
  – There are two kinds of nodes:
    » each sort in an operation is a node.
    » Each operation is also represented by a node.

• There are two kinds of arcs.
  
  – Indirected arcs from argument sorts nodes to an operation node,
  – and a directed node from an operation to the resulting sort.
Algebraic specifications: formal definitions

\[ f : s_1 \ldots s_n \rightarrow s \]
Algebraic specifications : formal definitions

- **Examples:**
- (a) nat
  
<table>
<thead>
<tr>
<th>Sorts</th>
<th>nat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ops</td>
<td>0 : $\rightarrow$ nat</td>
</tr>
<tr>
<td></td>
<td><code>succ : nat $\rightarrow$ nat</code></td>
</tr>
</tbody>
</table>

```
succ
```

```
nat
- 0
```

Rigoros and Adaptive ... Information Systems 23
Algebraic specifications : formal definitions

- Examples:
- (b) bool
  
  Sorts: bool
  
  Ops: true, false: \( \rightarrow \) bool
  
  \( \neg: \text{bool} \rightarrow \text{bool} \)
  
  \( \land, \lor, \ldots: \text{bool} \times \text{bool} \rightarrow \text{bool} \)
Algebraic specifications : formal definitions

- Examples:
- (c) nat1 = bool + nat

\[ \text{Ops} \quad + : \text{nat} \times \text{nat} \to \text{nat} \]
\[ \leq : \text{nat} \times \text{nat} \to \text{bool} \]
**Algebraic specifications : formal definitions**

- **Examples:**
- (c) natstack = nat1 +

  - **Sorts** Stack
  - **Ops**
    - new : \( \rightarrow \) stack
    - push : stack \( \times \) nat \( \rightarrow \) stack
    - pop : stack \( \rightarrow \) stack
    - top : stack \( \rightarrow \) nat

```
Examples:
(c) natstack = nat1 +

Sorts   Stack
Ops     new : \( \rightarrow \) stack
        push : stack \( \times \) nat \( \rightarrow \) stack
        pop : stack \( \rightarrow \) stack
        top : stack \( \rightarrow \) nat
```

![Diagram of natstack operations](image)
Examples:

(c) natqueue = nat1 +

Sorts
queue

Ops
empty : \rightarrow queue
in : queue × nat \rightarrow queue
out : queue \rightarrow queue
front : queue \rightarrow nat
Algebraic specifications: formal definitions

• There is a close relationship between signatures $\Sigma \langle S, F \rangle$ and context-free grammars.

$$G(\Sigma) = \{G(\Sigma)_{s_0} = (N,T,s_0,P) \} \quad s_0 \in S$$

- Non terminal symbols are $N = S$
- The terminal symbols are $T = F \cup \{(,}\} \cup \{,\}$ where $F$ is the disjunctive union of sets $f_{s,s}$
- The start symbol for $G(\Sigma)_{s_0}$ is $s_0$.
- The production rules are given by:

$$P = \{ s \rightarrow f(s_1, \ldots, s_n) / f \in F \}$$

$G(\text{bool}) = \{ \text{bool} \rightarrow \text{true} | \text{false} | \neg \text{bool} | \text{bool} \land \text{bool} | \ldots\}$
Algebraic specifications: formal definitions

• Definition 2 (Terms): given a signature \( \Sigma = \langle S, F \rangle \), With \( S \) a set of sorts, and \( F = \{f_{s_1, \ldots, s_n, s}\} \).

• A set of (closed) terms of a sort \( s \) over \( F \), denoted \( T_{F,s} \) are constructed as follows:
  1. Each constant of sort \( s \) is a term in \( T_{F,s} \).
  2. If \( t_1, \ldots, t_n \) are terms of sorts \( s_1, \ldots, s_n \) respectively and \( f:s_1,\ldots,s_n,s \) is in \( F \) then \( f(t_1,\ldots,t_n) \) is a term in \( T_{F,s} \).
  3. There no terms of sort \( s \) except those from 1. and 2.
**Algebraic specifications : formal definitions**

- **Examples:**
  - Nat spec. : \( 0 ; \text{succ}(0) ; \text{succ}(%28\text{succ}(0)\%29) ; ....; \)
    \(+\text{succ}(\text{succ}(0),0) ; \text{succ}(+(%28\text{succ}(0),\text{succ}(\text{succ}(\text{succ}(0)))))\).
  - bool spec. : \( \text{true} ; \neg\text{true} ; \land(%28\text{true}, \neg\text{true}\%29) ; ....\)
  - Natstack spec. : \( \text{new} ; \)
    \( \text{push}(\text{push}(\text{push}(\text{new},0),0),\text{succ}(0)); \)
    \( \text{push}(\text{pop}(\text{new}),\text{top}(\text{pop}(\text{new}))); ....\)

- **In addition of this prefix notations of terms, it is also possible to use the usual infix, postfix or mixfix notations :**
  \( \text{succ}(0) + \text{succ}(\text{succ}(0)); \text{succ}(\text{succ}(0))!, \text{if} \text{true} \text{then} \text{succ}(0) \text{else} \text{succ}(\text{succ}(\text{succ}(0))) \)}
Algebraic specifications : formal definitions

• **Terms with variables**: usually to express and use terms we include in terms also **variables**.

• Given a signature \( \text{SIG} = \langle S, F \rangle \), we generally introduce a set \( X_s \) called a **set of variables of sort** \( s \). Such variables are assumed to be pairwise disjoint and also disjoint with \( F \). The union \( X = \bigcup_{s \in S} X_s \) is called **set of variables w.r.t.** \( \text{SIG} \).

• **Terms (with variables) of sort** \( s \) are constructed as in definitions 2, with additionally in **1. variables of sort** \( s \) are also terms.
Algebraic specifications: formal definitions

• Examples:
  - Nat spec. : $X_{nat} = \{x, y\}$  $T_{nat}(X) = \{0; \text{succ}(0); \text{succ}(\text{succ}(x)); \ldots; +(\text{succ}(\text{succ}(0)),y); \text{succ}(+(\text{succ}(x),\text{succ}(\text{succ}(\text{succ}(x)))))) \ldots \}$
  - bool spec. : $X_{bool} = \{z\}$  $z, \text{true} ; \lnot\text{true} ; \land(z, \lnot\text{true}) ; \ldots$
  - Natstack spec. : $X_{stack} = \{st\}$  new ; push(push(push(st,0),0),\text{succ}(0)) ; push(pop(new),top(pop(st))) ; \ldots

• Terms without variables are called ground or closed terms. 0 ; succ(0) ; ...
Definition 3: An algebra \( A = (S^A, F^A) \) of a signature \( \text{SIG}= \langle S, F \rangle \), also called SIG-algebra, is given by two families \( S^A = (A^s)_{s \in S} \) and \( F_A = (f^A)_{f \in F} \), where

1. \( A_s \) are sets for all \( s \in S \), called base sets or domains of \( A \).
2. \( f^A \) are elements \( f^A \in A^s \) for all constant symbols in \( F \) i.e.
3. \( f : \rightarrow s \) and \( s \in S \), called constant.
4. \( f^A : A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n} \rightarrow A_s \) are functions for all operation symbols \( f : s_1 \ldots s_n, s \) from \( F \), called operations of \( A \), where „\( \times \)‟ denotes the cartesian product of sets.

If a signature \( \text{SIG}= \langle S, F \rangle \) is given as a list \( s_1, \ldots, s_n \) of sorts and a list \( f_1, \ldots, f_n \) of constants and operation symbols a SIG-algebra is represented as a list
\[ A = (A^{s_1}, \ldots, A^{s_n}, f_1^A, \ldots, f_n^A) \]
Algebraic specifications: formal definitions

- Example:
  
  Nat-algebra: $\text{NAT} = (S_{\text{nat}}, F_{\text{nat}}) = (\text{nat}^{\text{nat}}, 0^{\text{nat}}, \text{succ}^{\text{nat}})$

  » $\text{nat}^{\text{nat}} = \{0, 1, 2, 3, \ldots, n, \ldots\} = \text{naturals}$;
  
  » $0^{\text{nat}} = 0$ (of the natural);
  
  » $\text{succ}^{\text{nat}} = \{\text{successor (of the naturals)}: \text{naturals} \rightarrow \text{naturals}\}$. 
Algebraic specifications : formal definitions

- **Binary trees**:
  
  Binarytree-base =
  
  sorts : alphabet
  bintree
  
  opns : K1,...,Kn : alphabet
  
  LEAF : alphabet \rightarrow bintree
  RIGHT : bintree \rightarrow alphabet \rightarrow bintree
  BOTH : bintree \rightarrow alphabet \rightarrow bintree \rightarrow bintree

  BINTREE-BASE = (A, B, a1,.., an, leaf, left, right, both); where
  
  - A = \{a1,..,an\}; B : the set of binary trees
  - leaf : A \rightarrow B is defined by leaf(ai) = tree(ai) for i = 1,..,n
  - left : B \times A \rightarrow B adds a left subtree ;
  - right : A \times B \rightarrow B adds a right subtree, and
  - both : B \times A \times B \rightarrow B adds a left and a right subtree .
• Define the terms of bintree, and give the term corresponding to:

```
    K1
   /  \
K2   K4
  /   /  \
K3 K5 K6
```
Now we are going to define the evaluation of terms with and without variables in a given algebra $A$.

For terms with variables we have to start an assignment for the variables.

**Definition 4 (evaluation of terms):**

1. Let $T_F$ be the set of terms of a signature $\text{SIG}= \langle S, F \rangle$ and $A$ a $\text{SIG}$-algebra. The evaluation $\text{eval} : T_F \rightarrow A$ is recursively defined by:
   
   (i) $\text{Eval}(f) = f^A$ for all constant symbols $f \in F$
   (ii) $\text{Eval}(f(t_1, \ldots, t_n)) = f^A(\text{eval}(t_1), \ldots, \text{eval}(t_n))$ for all terms $f(t_1, \ldots, t_n) \in T_F$
2. Given a set of variables $X$ and $\text{SIG} = (S,F)$ and an assignment $\text{ass} : X \rightarrow A$ with $\text{ass}(x) \in A_s$ for $x \in X_s$ and $s \in S$. The extended assignment, or simply extension $\overline{\text{ass}} : T_F(X) \rightarrow A$ of the assignment $\text{ass} : X \rightarrow A$ is recursively defined by:

(i) $\overline{\text{ass}}(x) = \text{ass}(x)$ for all variables $x$
(ii) $\overline{\text{ass}}(f(t_1,...,t_n)) = f^A(\overline{\text{ass}}(t_1),...,\overline{\text{ass}}(t_n))$ for all $f(t_1,...,t_n) \in T_F(X)$. 

Algebraic specifications : formal definitions
• For $X = \emptyset$ there is exactly one assignment $\text{ass}$ which the "empty assignment", and we have $\text{ass} = \text{eval}$.

• We have then families of functions:
  - $\text{eval} = (\text{eval}_s : T_{F,s} \to A^s)_{s \in S'}$
  - $\text{ass} = (\text{ass}_s : X_s \to A^s)_{s \in S}$ and
  - $\overline{\text{ass}} = (\overline{\text{ass}}_s : T_{F,s}(X) \to A^s)_{s \in S}$

• The diagrams (1) and (2) commute
  
  » That is $T_{F,s} \to T_F(X) \xrightarrow{\text{ass}} A = T_F \xrightarrow{\text{eval}} A$
  » and $X \to T_F(X) \xrightarrow{\text{ass}} A = X \xrightarrow{\text{ass}} A$
Algebraic specifications: formal definitions

• Evaluate $\text{add}(\text{succ}(n),m)$, with $X_{\text{nat}} = \{n,m\}$ and $\text{Assx}(n)=5$ and $\text{assx}(m) = 3$
By choosing an algebraic semantics to abstract data types, algebraic structures (e.g. monoid, group, rings, ..) are also endowed with equations

» They express the relationship between different operators.

» They allow to describe complex (defined) function symbols using elementary (constructor) operation symbols.

» They allow us to have a simplified form of any term.
**Algebraic specifications : Equations**

- **Examples:**
  - (a) bool

  **Sorts**  bool
  **Ops**  true, false : → bool
            ¬ : bool  → bool
            ∧, ∨ : bool x bool  → bool
  **if_then_else Fi** : bool x bool x bool  →  bool

  **Vars**  p, q : bool
  **Eqs**  if true then p else q fi = p
           if false then p else q fi = q
           ¬ p =if p then false else true fi
           p ∧ q =if p then q else false fi
           p ∨ q =if p then true else q fi
(b) nat =

Sorts  nat
Ops  0 : \rightarrow bool
     succ : nat \rightarrow nat
     add : nat \times nat \rightarrow nat
Vars  n, m : nat
Eq  add(n, 0) = 0
    add(succ(n), m)) = succ(add(n,m))
Algebraic specifications: Equations

- (b) \text{nat1} = \text{nat} + \text{bool} +

\text{Ops} \quad \leq : \text{nat} \times \text{nat} \rightarrow \text{bool}

\text{Vars} \quad n, m : \text{nat}

\text{Eqs} \quad (0 \leq n) = \text{true}

\quad (\text{succ}(n) \leq 0) = \text{false}

\quad (\text{succ}(n) \leq \text{succ}(m)) = (n \leq m)
Algebraic specifications : Equations

- (b) natstack = nat1+

Sorts  S

Ops  new : \rightarrow S
     push : S \times \text{nat} \rightarrow S
     pop : S \rightarrow S
     top : S \rightarrow \text{nat}

Vars  s : S ; n : \text{nat}

Eqs  pop(push(s,n)) = s
     top(push(s,n)) = n
     pop(new) = new
     top(new) = 0
**Algebraic specifications : Equations**

- **Definition**: Given a signature \( \text{SIG}=\langle S, F \rangle \) and variables \( X \) w.r.t. \( \text{SIG} \)
  1. A triple \( e=(X, L, R) \) with \( L, R \in T_{F,s}(X) \) for some \( s \in S \), is called an equation of sort \( s \) w.r.t. \( \text{SIG} \).
  2. The equation \( e=(X, L, R) \) is called valid in a \( \text{SIG} \)-algebra \( A \) if for all assignments \( \text{ass}: X \rightarrow A \) we have \( \text{ass}(L) = \text{ass}(R) \).
    Where \( \text{ass} \) is the extended assignment of \( \text{ass} \). If \( e \) is valid in \( A \) we also say that \( A \) satisfies \( e \).
  3. Ground equations are equations \( e=(X, L, R) \) with \( X = \emptyset \) (that is when \( L \) and \( R \) are ground terms).
An equation is a universal first order formula

- $\forall x_1 \in s_1, \forall x_2 \in s_2 \ldots \forall x_n \in s_n \ (L = R)$ written usually as: $x_1 \in s_1, x_2 \in s_2 \ldots x_n \in s_n \ (L = R)$

- X must contains all variables occurring in L and R.

- In general, for sake of simplicity the variables set is omitted.
Algebraic specifications: Specification and SPEC-algebra

• **Definition (specification and SPEC-algebra):**
  1. A specification SPEC = ⟨S, F, E⟩ consists of a signature SIG = ⟨S, F⟩ and a set E of equations e w.r.t. SIG variables X w.r.t. SIG.
  2. An algebra A of the specification SPEC, short SPEC-algebra, is an algebra A of the signature SIG which satisfies all equations in E.
Algebraic specifications: Specification and SPEC-algebra

- A specification is also called algebraic specification or equational specification.
- If a specification SPEC1 consists of a given specification SPEC and additional sorts S1, operations F1, and equations E1, we write this in the form:

  $$\text{SPEC1} = \text{SPEC} + (S1, F1, E1)$$

  Which means

  $$\text{SPEC1} = (S, S1, F + F1, E, E1)$$
• Show that \( \text{NAT} = (\mathbb{N}, =, +1, +) \) is a nat-algebra, that is for each assignment \( \text{ass}: X \rightarrow \mathbb{N} \) with \( X = \{n, m\} \) the extended assignments applied to both equations deliver the same values.
**Algebraic specifications : Specification and SPEC-algebra**

- **Definition (derivation of rewriting of terms):**
  Given a set $E$ of equations for a signature with a fixed set of variables $X = X_e$ for each equation $e$. $(L,R) \in E$ defines two substitution rules:
  
  1. $L \Rightarrow R$ (L-R-rule)
  2. $R \Rightarrow L$ (R-L-rule)

  A rule $t_1 \Rightarrow t_2$ is applicable to a term $t \in T_F(X)$ if there is an assignment $\text{ass}: X \rightarrow T_F(X)$ with extension $\text{ass} : T_F(X) \rightarrow T_F(X)$ such that we have for $t_1 = \text{ass}(t_1)$ and $t_2 = \text{ass}(t_2)$:

  3. $t_1$ is a subterm of $t$. 
Algebraic specifications: Specification and SPEC-algebra

The replacement of \( t_1 \) in \( t \) by \( t_2 \) yields a term \( t' \), the replacement of \( t_1 \) by \( t_2 \) in \( t \) is denoted by:

\[
(4) \quad t' = t(t_1 / t_2)
\]

In this case we write

\[
(5) \quad t \Rightarrow t', \text{ called direct derivation from } t \text{ to } t' \text{ via } E \text{ using the rule } t_1 \Rightarrow t_2 \text{ and assignment ass.}
\]

\[
(6) \quad t \Rightarrow^* t \text{ represent any sequence } t_0 \Rightarrow t_1 \Rightarrow \ldots \Rightarrow t_n \text{ with } t = t_0 \text{ and } t' = t_n.
\]

It is called derivation from \( t \) to \( t' \) via \( E \) and it is correct w.r.t. SIG-algebra \( A \) if for each assignment \( \text{ass} \): \( X \Rightarrow A \)

\[
(7) \quad \text{ass}(t) \Rightarrow \text{ass}(t')
\]
Definition (occurrence or positions in terms):
- Given a term \( t \), the set of positions in \( t \), denoted by \( \text{Dom}(t) \), is the set of sequences of natural numbers defined as:
  - If \( t \) is constant or variable, then \( \text{Dom}(t) = \{\emptyset\} \)
  - If \( t \) is of the form \( f(t_1, \ldots, t_n) \) then
    \[
    \text{Dom}(t) = \{\emptyset\} \cup \{i.p \mid i \in \{1, \ldots, n\} \text{ and } p \in \text{Dom}(t_i)\}
    \]

Definition (subterms)
- Given a term \( t \), and a position \( p \in \text{Dom}(t) \) we define a subterm of \( t \) rooted at a position denoted \( t |_p \) as:
  - \( p = \emptyset \), then \( t |_p = t \)
  - If \( p = i.p' \) then \( t = f(t_1, \ldots, t_i, \ldots) |_{i.p'} = t_i |_{p'} \)
  - A term \( t' \) is said to be a subterm of \( t \) if there is a position \( p \) such that \( t' = t |_p \)
• **Definition (Term replacement)**
  - Given a term $t$, a position $p$, and a term $t'$, we define $t[p \leftarrow t']$ as
    » If $p = \emptyset$ then $t[p \leftarrow t'] = t'$
    » If $p = i.p'$ then $t = f(t_1, \ldots, t_{i-1}, t_i, t_{i+1})[i.p' \leftarrow t']
    = f(t_1, \ldots, t_{i-1}, t_i [p \leftarrow t'], t_{i+1})$

• **Definition (Rewriting term)**
  - Given a system of rules (oriented equations), $R$, we define a rewrite relation by $\Rightarrow_R$, as $t \Rightarrow_R t'$, if :
    » There is a rule $r : l \Rightarrow r$ is $R$ ; there is an assignment (substitution) $\sigma : X \rightarrow T_F(X)$ ; and a position $p$ in $t$ such that $t |_p = \sigma (l)$ and $t' = t[p \leftarrow \sigma (r)]$
Algebraic specifications: Specification and SPEC-algebra

**Definition (Congruence on Ground Terms):**
Given a specification $\text{SPEC} = (S, F, E)$ the relation $\equiv$ on ground terms defined for all $t_1, t_2 \in T_F$ by

$$t_1 \equiv t_2 \text{ if and only if } \text{eval}_A(t_1) = \text{eval}_A(t_2) \text{ for all SPEC-algebra A is called congruence on ground terms.}$$

It satisfies the following conditions for all $t_1, t_2, t_3 \in T_F$:
- $t_1 \equiv t_1$ (reflexivity); $t_1 \equiv t_2$ implies $t_2 \equiv t_1$ (symmetry);
- $t_1 \equiv t_2$ and $t_2 \equiv t_3$ implies $t_1 \equiv t_3$ (transitivity);
- $t_1 \equiv t_1',...,, t_n \equiv t_n'$ implies $f(t_1,...t_n) \equiv f(t_1',...,t_n')$ (congruence)
- each derivation $t_1 \Rightarrow t_2$ via $E$ between ground terms $t_1, t_2 \in T_F$ implies $t_1 \equiv t_2$.
A rewriting relation $\Rightarrow^R$ is like a congruence relation without the reflexivity property.
**Definition (Quotient Term Algebra $T_{SPEC}$):**

Given a specification $SPEC = (S, F, E)$ the quotient term algebra $T_{SPEC} = ((Q_s)_{s \in S}, (f_Q)_{f \in F})$ is defined by

1. For each $s \in S$, we have a base set
   
   $Q_s = \{[t] / t \in T_{F,s}\}$
   
   where the congruence class $[t]$ is defined by:
   
   $[t] = \{t' / t' \equiv t\}$

2. For each constant symbol $f : \rightarrow s$ in $F$ the constant $Q_s$ is the congruence class generated by $f$: $f_Q = [f]$

3. For each operation symbol $f:s1 \ldots sn \rightarrow s$ in $F$ the operation
   
   $f_Q : Q_{s1} \times \ldots \times Q_{sn} \rightarrow Q_s$ is defined by

   $f_Q([t1], \ldots,[tn]) = [f(t1,\ldots,tn)]$
Example (Quotient Term Algebra T_{nat}): 

\[ T_{nat} = (Q_{nat}, 0_Q, SUCC_Q, ADD_Q) \]

With

- \( Q_{nat} = \{[SUCC^n(0)] / n \geq 0 \} \)
- \( 0_Q = [0] \), and for \( n, m \geq 0 \)
- \( SUCC_Q([SUCC^n(0)]) = [SUCC^{n+1}(0)] \)
- \( ADD_Q([SUCC^n(0)], [SUCC^m(0)]) = [SUCC^{n+m}(0)] \)

Fact: \( T_{SPEC} \) is a SPEC-Algebra and it is called the initial semantics with \( \text{ADT}(\text{SPEC}) = \{ A / A \cong T_{SPEC} \} \) is called the (initial) abstract data type defined by SPEC.
**Definition (derivation of rewriting of terms):**

Given a set $E$ of equations for a signature with a fixed set of variables $X = X_e$ for each equation $e$. $(L,R) \in E$ defines two substitution rules:

1. $L \Rightarrow R$ (L-R-rule)
2. $R \Rightarrow L$ (R-L-rule)

A rule $t_1 \Rightarrow t_2$ is applicable to a term $t \in T_F(X)$ if there is an assignment $\text{ass}: X \rightarrow T_F(X)$ with extension $\text{ass} : T_F(X) \rightarrow T_F(X)$ such that we have for $t_1 = \text{ass}(t1)$ and $t_2 = \text{ass}(t2)$:

3. $t_1$ is a subterm of $t$. 
The replacement of \( t_1 \) in \( t \) by \( t_2 \) yields a term \( t' \), the replacement of \( t_1 \) by \( t_2 \) in \( t \) is denoted by:

(4) \[ t' = t(t_1 / t_2) \]

In this case we write

(5) \[ t \Rightarrow t', \] called direct derivation from \( t \) to \( t' \) via \( E \) using the rule \( t_1 \Rightarrow t_2 \) and assignment \( \text{ass} \).

(6) \[ t \Rightarrow^* t \] represent any sequence

\[ t_0 \Rightarrow t_1 \Rightarrow ... \Rightarrow t_n \] with \( t = t_0 \) and \( t' = t_n \).

It is called derivation from \( t \) to \( t' \) via \( E \) and it is correct w.r.t. SIG-algebra \( A \) if for each assignment \( \text{ass} \): \( X \rightarrow A \)

(7) \[ \overline{\text{ass}}(t) \Rightarrow \overline{\text{ass}}(t') \]
**Definition (occurrence or positions in terms):**
- Given a term $t$, the set of positions in $t$, denoted by $\text{Dom}(t)$, is the set of sequences of natural numbers defined as:
  » If $t$ is constant or variable, then $\text{Dom}(t) = \{\emptyset\}$
  » If $t$ is of the form $f(t_1, \ldots, t_n)$ then
    \[ \text{Dom}(t) = \{\emptyset\} \cup \{i.p / i \in \{1,\ldots,n\} \text{ and } p \in \text{Dom}(t_i)\} \]

**Definition (subterms)**
- Given a term $t$, and a position $p \in \text{Dom}(t)$ we define a subterm of $t$ rooted at a position denoted $t|_p$ as:
  » $p = \emptyset$, then $t|_p = t$
  » If $p = i.p'$ then $t = f(t_1, \ldots, t_{i'})|_{i.p'} = t_i|_{p'}$
  » A term $t'$ is said to be a subterm of $t$ is there is a position $p$ such that $t' = t|_p$
**Algebraic specifications : Specification and SPEC-algebra**

- **Definition (Term replacement)**
  - Given a term $t$, a position $p$, and a term $t'$, we define $t[p \leftarrow t']$ as
    - If $p = \emptyset$ then $t[p \leftarrow t'] = t'$
    - If $p = i.p'$ then $t = f(t_1, ..., t_{i-1}, t_i, t_{i+1}...)[i.p' \leftarrow t']$
      
- **Definition (Rewriting term)**
  - Given a system of rules (oriented equations), $R$, we define a rewrite relation by $\Rightarrow_R$, as $t \Rightarrow t'$, if:
    - There is a rule $r : l \Rightarrow r$ is $R$; there is an assignement (substitution) $\sigma : X \rightarrow T_F(X)$; and a position $p$ in $t$ such that $t|_p = \overline{\sigma}(l)$ and $t' = t[p \leftarrow \overline{\sigma}(r)]$
**Definition (Congruence on Ground Terms):**

Given a specification $\text{SPEC} = (S, F, E)$ the relation $\congruent{}$ on ground terms defined for all $t_1, t_2 \in T_F$ by

\[
t_1 \congruent{} t_2 \text{ if and only if } \text{eval}_A(t_1) = \text{eval}_A(t_2) \text{ for all SPEC-algebra } A
\]

is called congruence on ground terms.

It satisfies the following conditions for all $t_1, t_2, t_3 \in T_F$:

- $t_1 \congruent{} t_1$ (reflexivity) ; $t_1 \congruent{} t_2$ implies $t_2 \congruent{} t_1$ (symmetry) ;
- $t_1 \congruent{} t_2$ and $t_2 \congruent{} t_3$ implies $t_1 \congruent{} t_3$ (transitivity) ;
- $t_1 \congruent{} t_1^\prime ,..., t_n \congruent{} t_n^\prime$ implies $f(t_1,..,t_n) \congruent{} f(t_1^\prime,..,t_n^\prime)$ (congruence)
- each derivation $t_1 \rightarrow t_2$ via $E$ between ground terms $t_1, t_2 \in T_F$ implies $t_1 \congruent{} t_2$. 
Algebraic specifications : Specification and SPEC-algebra

- A rewriting relation $\Rightarrow_R$ is like a congruence relation without the reflexivity property.

- $\text{Top}(\text{push}(\text{pop}(\text{push}(\text{empty},0)), \text{succ}(m)))$
  $= \text{top}(\text{push}(\text{empty}, \text{succ}(m)))$
  $= \text{succ}(m)$
Definition (Algebra of Terms)

Given a signature $\text{SIG} = (S, F)$. We define the algebra of terms $T = (S^T, F^T)$ w.r.t. SIG and a set of variables $X$ or simply term algebra as:

1. $S^T = \{ (T_{F,s}(X))_{s \in S} \}$ as the family of base sets
2. $f^T := f$ as the constant for $f : \rightarrow s$
3. $f^T : T_{F,s_1}(X) \times \ldots \times T_{F,s_n}(X) \rightarrow T_{F,s}(X)$ defined by $f^T(t_1,\ldots,t_n) := f(t_1,\ldots,t_n)$ for $f : s_1 \ldots s_n \rightarrow s$ and $t_i \in T_{F,s_i}(X)$
**Definition (Quotient Term Algebra $T_{SPEC}$):**

Given a specification $SPEC = (S, F, E)$ the quotient term algebra $T_{SPEC} = ((Q_s)_{s \in S}, (f_Q)_{f \in F})$ is defined by

1. For each $s \in S$, we have a base set
   
   $$Q_s = \{[t] / t \in T_{F,s}\}$$

   where the congruence class $[t]$ is defined by:
   
   $$[t] = \{t' / t' \cong t\}$$

2. For each constant symbol $f : \rightarrow s$ in $F$ the constant $Q_s$ is the congruence class generated by $f$: $f_Q = [f]$.

3. For each operation symbol $f: s_1 \ldots s_n \rightarrow s$ in $F$ the operation
   
   $$f_Q : Q_{s_1} \times \ldots \times Q_{sn} \rightarrow Q_s$$

   is defined by
   
   $$f_Q([t_1], \ldots, [t_n]) = [f(t_1,\ldots,t_n)]$$
Example (Quotient Term Algebra $T_{\text{nat}}$):

$T_{\text{nat}} = (Q_{\text{nat}}, 0_Q, \text{SUCC}_Q, \text{ADD}_Q)$

With

- $Q_{\text{nat}} = \{\text{SUCC}^n(0) / n \geq 0\}$
- $0_Q = [0]$, and for $n, m \geq 0$
- $\text{SUCC}_Q([\text{SUCC}^n(0)]) = [\text{SUCC}^{n+1}(0)]$
- $\text{ADD}_Q([\text{SUCC}^n(0)], [\text{SUCC}^m(0)]) = [\text{SUCC}^{n+m}(0)]$

Fact: $T_{\text{SPEC}}$ is a SPEC-Algebra and it is called the initial semantics with $\text{ADT}(\text{SPEC}) = \{A / A \cong T_{\text{SPEC}}\}$ is called the (initial) abstract data type defined by SPEC.
Algebraic specifications: Specification

- The quotient term algebra $T_{SPEC}$ of a specification $SPEC = (S, F, E)$ has the following properties:
  1. The evaluation $eval : T_F \rightarrow T_{SPEC}$ is equal to $nat : T_F \rightarrow T_{SPEC}$, defined by $nat(t) = [t]$ for all $t \in T_F$, and hence surjective.
  1. Each equation $e = (t_1, t_2)$ of ground term $t_1, t_2 \in T_F$ is valid in $T_{SPEC}$ if and only if it is valid in each $SPEC$-algebra $A$.
  1. $T_{SPEC}$ is a $SPEC$-algebra.
**Algebraic specifications : Specification**

- **Definitions (Equational Rules and Proofs)**
  1. An equational rule (over SIG) is given by a pair \((E, e)\)
     
     Where \(E\) is a set of equations and \(e\) is a single equation w.r.t. SIG. We also write \(E \mid-- e\)
  2. Given a set \(R\) of equational rules and a set of \(E\) of equations w.r.t. SIG. Then an (equational) proof
     
     With rules \(R\) and premisses \(E\) is a sequence \(E\) is a sequence \(e_1, \ldots, e_r\).
Algebraic specifications : Specification

- **Definition (Equational calculus)**: The equational calculus is defined to contain exactly the following equational rules:

  for $t_1, t_2, t_3 \in T_F(X)$ and $t \in T_F(Y)$.

  - **R1**: $ |-- t_1 = t_1$ (identity)
  - **R2**: $t_1 = t_2$ $ |-- t_2 = t_1$ (symmetry)
  - **R3**: $t_1 = t_2$ and $t_2 = t_3$ $ |-- t_1 = t_3$ (transitivity)
  - **R4**: $(X, t_1 = t_2)$ $ |-- (X \cup Y, h(t_1) = h(t_2))$ for ass: $X \rightarrow T_F(Y)$ (substitivity).
**Algebraic specifications : Specification**

- When the equations are used as rewrite rules, the symmetric rule is to be dropped.
- In rewriting techniques, the process of orienting equations is based on the so-called simplication orderings: a partial order between operations extended to terms.
  - example: \( \text{add} > \text{succ} > 0 \Rightarrow \text{add}(\ldots,\ldots) > \text{succ}(\ldots) \)
- In order to ensure the termination of rewriting of a term, such ordering is required to be well-founded (any ordering should have has a small element).
- The small element a any term is called the normal form of the term.
In order to ensure the uniqueness of computation, the so-called confluence property is required:

- $t_1 \Rightarrow^* t_2 \Rightarrow^* Nf(t_1)$ and
- $t_1 \Rightarrow^* t_3$
- then $t_3$ should be rewritten to $Nf(t_1)$ i.e. $t_3 \Rightarrow^* Nf(t_1)$

This property is ensured by the so-called Knuth-Bendix completion procedure. It takes a set of equations and an ordering, and it generate a set of rewrite rules which terminate and are confluent.
The confluence property is verified by eliminating all ambiguities that may be hidden between different rules of the system. These ambiguities are called critical-pairs.

For their definition, we need the notion of unification:

Two terms $t_1$ and $t_2$ are said to be unifiable if there is a substitution $\sigma$ such that $\sigma(t_1) = \sigma(t_2)$.

Example: let $t_1 = f(a, g(y))$ and $t_2 = f(x, g(h(b)))$, then it is easy to prove that $\sigma = \{x \rightarrow a, y \rightarrow h(b)\}$ is a unifier of $t_1$ and $t_2$. That is, $\sigma(t_1) = f(a, g(h(b))) = \sigma(t_2)$.
Algebraic specifications : Specification

- **Definition (critical pair)**
  If \( l \rightarrow r \) and \( s \rightarrow t \) are two rewrite rules with distinct variables, \( p \) is the position of a nonvariable subterm of \( s \), and \( \mu \) is the unifier of \( s|_p \) and \( l \), then the equation \( \mu(t) = \mu(s[\mu(r)]|_p) \) is a critical pair formed from those rules.

- **Example**: suppose we want to add the alternation in the stack specification using the following two rewrite rules:
  \[
  \text{alternate}(\text{push}(x,y),z) \rightarrow \text{push}(x, \text{alternate}(z,y))
  \]
  \[
  \text{Alternate}(y_1, \emptyset) \rightarrow y_1
  \]
  Then, by applying the above definition, we can notice that \text{alternate} in the second rule occurs at position \( \emptyset \) in the first rule. That is, \( s = \text{alternate}(\text{push}(x,y),z) \) and \( l = \text{alternate}(y_1, \emptyset) \). So, we have to check for a unification of \( \text{alternate}(z,y) \) and \( \text{Alternate}(y, \emptyset) \). The unifier here is \( \mu = \{y_1 \rightarrow \text{push}(x,y), z \rightarrow \emptyset\} \)
Algebraic specifications: Specification

• So, the resulting members of the critical pair are
  1. \( \mu(t) = \mu(\text{push}(x, \text{alternate}(z,y))) = \text{push}(x,\text{alternate}(\emptyset,y)) \)
  
  2. \( \mu(t) = \mu(s[\mu(r)] | p) = \mu(s[\mu(r)] | \emptyset) = \mu(r) = \mu(y1) = \text{push}(x,y) \)

• And the critical pair is therefore the resulting equation:
  \( \text{push}(x, \text{alternate}(\emptyset, y)) = \text{push}(x,y) \)
Advanced Algebraic specifications

To deal a maximal of cases and errors, *subsorts* may be defined. \( S < S' \)

**Parametrized specifications** are specifications based on others: stack(string); list(nat); ....

- **Parametrized specifications** are interpreted using category on algebras.

- **To go beyond the non-changing or fixed notions of algebras**, and thereby interpreting state-based reactive (information) systems, several *extensions* have been proposed to the algebraic semantics.
  - **Rewriting logic**: a computation is a functor from an algebra to another.
  - **Hidden sorted algebra**: some sorts modelling states are hidden.