The Reference Manual for the SMoLCS Methodology
Version 4.2 – 2/20/95*

G. Reggio – D. Bertello – A. Morgavi
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova – Italy
email smolcs@disi.unige.it

SMoLCS is a formal algebraic specification methodology for dynamic systems, i.e. for systems evolving along the time. This reference manual for SMoLCS is intended for users of different training as could be found in an industrial environment; thus it is "self-sufficient", i.e. it contains all notions needed for understanding such methodology. The principles and the fundamental ideas of SMoLCS are formally introduced in [?, ?, ?, ?, ?, ?, ?], here such ideas are presented in a more immediate form trying to isolate the most formal parts.

In the first section we introduce the algebraic specifications of data type, starting from the concept of concrete data type formalized as algebra on a signature, until to the specification at different levels of abstraction throughout the formalization always more precise of the properties of the data type.

In the second section we illustrate the use of labelled transition systems for modelling dynamic systems, also structured ones.

In the third section we introduce the specifications of dynamic systems, starting from the dynamic algebras integrating the concept of abstract data type with that of labelled transition system, using also temporal combinators for formalizing abstract properties on the activity of the systems and the concept of entity for formalizing the structural properties of dynamic systems.

The document is organized as follows:

- the fundamental concepts are introduced using simple examples, for allowing a more easy understanding;
- the precise definitions formalizing them are reported in figures, for isolating them in the text and for allowing to exclude them at a first reading, moreover

*This work has been supported by a grant ENEL/CRA (Milano Italy) and by "Progetto Finalizzato SistemiInformatici e Calcolo Parallelo" of C.N.R. (Italy).

- there are more complex complete examples, for clarifying and further depthening such concepts.

The examples are given using the specification language associated with the methodology presented in [**?**].

Mathematical notations In the following we briefly report the main mathematical notations used in this manual.

- Let A and B be two sets:
  - $a \in A$ means that a belongs to the set A;
  - $A \cup B$ denotes the union between the set A and the set B, i.e. the set whose elements belong either to A or to B;
  - $A \times B$ denotes the cartesian product of the two sets in such order, i.e. the set of the pairs (a, b) s.t. $a \in A$ and $b \in B$;
  - A^* denotes the set of all streams (finite sequences) on A, i.e., the set $\cup_{n \geq 0} A^n$, where $A^0 = \{\Lambda\}$ (Λ is said empty stream) and for $n \geq 1$ $A^n = A \times \ldots \times A$, n times; moreover the set of nonempty streams $A^* - \{\Lambda\}$ is denoted by A^+ and the stream concatenation is denoted by "$\cdot$";
  - $S(A)$ denotes the set of the finite sets of elements of A;
  - $M(A)$ denotes the set of the finite multisets of elements of A, i.e., finite sets with possible multiple occurrences of the same element (formally defined by means of a molteplicity function). For example, $\{1, 1, 4, 1, 4\}$ is a multiset with 3 occurrences of 1 and 2 occurrences of 4 and is equal to $\{1, 1, 1, 4, 4\}$;
  - $(A \to B)$ [$(A \to B)_{partial}$] denotes the set of the total [partial] functions from A into B.

- $F: A_1 \times \ldots \times A_n \to B$

  $F(x_1, \ldots, x_n) = e$

  denotes the function from the cartesian product of the sets $A_1, \ldots, A_n$ into the set B, defined by the law associating with each tupla $x_1, \ldots, x_n$ the value represented by the expression e.

  Notice that it is possible to denote also functions with mixfix syntax; for example the infix binary function "@" is denoted by $\_@\_: A_1 \times A_2 \to B$ and its association law is given in the form $x_1 @ x_2 = e$.

- $\mathbb{N}$ denotes the set of the natural numbers (with their structure, when it is needed).

- $\wedge$, $\vee$, $\neg$, $\supset$ denote the logic combinators: conjunction ($b_1 \wedge b_2$ is true whether both b_1 and b_2 are true), nonexclusive disjunction ($b_1 \vee b_2$ is true whether either b_1 or b_2 is true, also both), negation ($\neg\, b$ is true whether b is false) and implication ($b_1 \supset b_2$ is true whether either b_1 is false or b_2 is true).

1 Algebraic specifications of data types

1.1 Concrete data types as algebras

A *concrete data type* consists of a family of sets, one of constants, one of functions and one of predicates (functions returning truth values) on such sets; recall, indeed, that a concrete data type is not defined only by the sets of elements composing it, but also describing how such elements are handled. Moreover recall also that not all handlings of the elements of a data type are admissible, some have as result "non-good" elements, corresponding to errors; an immediate example for the naturals is the division by zero.

Consider now the well-known case of the stack of naturals; it is not sufficient to define the set of the stacks as the set of all streams of naturals, we also need to specify as such streams are handled for characterizing a stack. Indeed, if we specify that the elements of the streams, the naturals, are put and taken from the head of the streams we get the data type concrete stack, otherwise it is possible to get different data types, as the queue in the case we put the elements on the head of the streams and take them from the tail.

A concrete way for defining the data type stack is to consider it as made by:

- the sets
  - $\mathbb{N} \cup \{\#nat\}$, the natural numbers together a symbol corresponding to the error for naturals;
  - $\mathbb{N}^* \cup \{\#stack\}$, the streams of naturals together a symbol corresponding to the error for the stacks;

  (notice that here and in the following the elements corresponding to"errors" are denoted by symbols starting with "#")
- the constants
  - $0 \in \mathbb{N} \cup \{\#nat\}$;
  - $\Lambda \in \mathbb{N}^* \cup \{\#stack\}$, the empty stack;
- the functions
  - $SUCC \colon \mathbb{N} \cup \{\#nat\} \to \mathbb{N} \cup \{\#nat\}$ (successor)
    $SUCC(n) = \#nat$ if $n = \#nat$, $n+1$ otherwise;
  - $\_+\colon \mathbb{N} \cup \{\#nat\} \times \mathbb{N} \cup \{\#nat\} \to \mathbb{N} \cup \{\#nat\}$ (addition)
    $n + m = \#nat$ if either $n = \#nat$ or $m = \#nat$, $n + m$ otherwise;
  - $PUT \colon \mathbb{N} \cup \{\#nat\} \times \mathbb{N}^* \cup \{\#stack\} \to \mathbb{N}^* \cup \{\#stack\}$ (puts a natural on a stack)
    $PUT(n, s) = \#stack$ if either $n = \#nat$ or $s = \#stack$, $n \cdot s$ otherwise;
  - $GET \colon \mathbb{N}^* \cup \{\#stack\} \to \mathbb{N}^* \cup \{\#stack\}$ (takes away from a stack the element on the top)
    $GET(s) = \#stack$ if either $s = \#stack$ or $s = \Lambda$, s' if $s = n \cdot s'$;

  - $FIRST\colon \mathbb{N}^* \cup \{\#stack\} \rightarrow \mathbb{N} \cup \{\#nat\}$ (returns the element on the top of a stack) $FIRST(s) = \#nat$ if either $s = \#stack$ or $s = \Lambda$, n if $s = n \cdot s'$;

- and the predicates
  - $IS\_EMPTY\colon \mathbb{N}^* \cup \{\#stack\} \rightarrow \{TRUE, FALSE\}$ (checks whether a stack is empty)
    $IS\_EMPTY(s) = TRUE$ if $s = \Lambda$, $FALSE$ otherwise;
  - $OK_{nat}\colon \mathbb{N} \cup \{\#nat\} \rightarrow \{TRUE, FALSE\}$ (determines the "good" values of the naturals, i.e. those that do not correspond to an "error")
    $OK_{nat}(n) = FALSE$ if $n = \#nat$, $TRUE$ otherwise
  - $OK_{stack}\colon \mathbb{N}^* \cup \{\#stack\} \rightarrow \{TRUE, FALSE\}$ (determines the "good" values of the stacks)
    $OK_{stack}(s) = FALSE$ if $s = \#stack$, $TRUE$ otherwise.

The above definition of the concrete data type stack may be given more precisely as an algebra, i.e. a pair consisting of a signature and of the interpretation of the symbols appearing in the signature; the signature gives in a certain sense the syntax for representing and handling the values of the type (i.e., the symbols of the different kinds of values of the type and the symbols of the constants, operations and predicates of the type) while the interpretations associated with such symbols (sets, elements, functions and predicates) are the components of the data type.

1.1.1 Signatures

Formally a *signature* is a 4-uple made by four sets of symbols said respectively *sorts*, *constants*, *operations* and *predicates*. Each constant symbol has associated an arity (the symbol of the type of the constant) and we write $Cn\colon s$ for denoting that Cn has arity s. Each operation symbol has associated an arity (a pair made by a stream of sorts [the symbols of the types of the arguments] and by a sort [the symbol of the type of the result]) and we write $Op\colon s_1 \; \ldots \; s_n \rightarrow s$ for denoting that Op has arity $(s_1 \ldots s_n, s)$. Each predicate symbol has associated an arity (a stream di sorts, the symbols of the types of the arguments) and we write $Pr\colon s_1 \; \ldots \; s_n$ for denoting that Pr has arity $s_1 \ldots s_n$; indeed a predicate symbol represents a function returning boolean a values of whom we consider implicitly the result sort.

We require that all signatures for each sort s include a special predicate symbol Ok_s which determines the "good" elements of sort s.

The formal definition of signature is reported in Fig. 1.

For example the signature, `S_Stack`, for the data type stack expressed using the specification language associated to our methodology METAL (see [?]) is the following:

```
sortnat stack
cn  Zero:  nat
cn  Empty:  stack
op  Succ:  nat -> nat
```

A *signature* is a 4-uple $\Sigma = (S, CN, OP, PR)$, where

- S is a set, the sorts;
- $CN = \{CN_s\}_{s \in S}$ are the constant symbols (the elements of CN_s are the symbols of the constants with arity s);
- $OP = \{OP_{w,s}\}_{w \in S^+, s \in S}$ are the symbols of the operations (the elements of $OP_{w,s}$, are the operation symbols with arity (w, s));
- $PR = \{PR_w\}_{w \in S^+}$ are the predicates symbols (the elements of PR_w are the symbols of the predicates with arity w) s.t. $Ok_s \in PR_s$ for each $s \in S$.

Figure 1: Signature.

```
op  _ + _ :  nat nat -> nat     **   operation with mixfix syntax
op  Put:  nat stack -> stack
op  Get:  stack -> stack
op  First:  stack -> nat
pr  Is_Empty:  stack
pr  OKnat:  nat
pr  OKstack:  stack
```

1.1.2 Algebras

An *algebra* on a signature is a family of interpretations of the symbols of sorts, constants, operations and predicates of the signature. The interpretation of a symbol of sort is a set, said the *carrier* associated with the sort; the interpretation of a constant symbol of arity s is an element of the carrier associated with the sort s; the interpretation of an operation symbol is a function between the carriers associated with the sorts of the arguments and that of the result; analogously the interpretation of a predicate symbol is a function between the carriers associated with the sorts of the arguments and the set of the truth values $\{TRUE, FALSE\}$ and is representable also with a subset of the cartesian product of the carriers associated with the sorts of the arguments.

In the case of the stacks we call STACK the algebra on the signature `S_Stack` in which the carriers associated with the sorts `nat` and `stack` are respectively the sets $\mathbb{N} \cup \{\#nat\}$ and $\mathbb{N}^* \cup \{\#stack\}$; the interpretation of the constant symbols `Zero` and `Empty` are respectively the natural 0 and the empty stack Λ; the interpretation of the operation symbols `Put`, `Get`, `First` are respectively the functions PUT, GET, $FIRST$ and the interpretation of the predicate symbols `Is_Empty`, `OKnat`, `OKstack` are respectively IS_EMPTY, OK_{nat}, OK_{stack}.

The formal definition of algebra is reported in Fig. 2.

Notice that the conditions which must be satisfied by the elements of the carriers of an algebra A on a signature Σ correspond to our ideas about the errors, in particular we want that there is only one element corresponding to error for each carrier, that the constants are always "good" and that the functions and the predicates are "strict". More in details:

Given a signature $\Sigma = (S, CN, OP, PR)$, an *algebra* on Σ, or Σ-*algebra*, is a 4-uple $\mathrm{A} = (\{s^{\mathrm{A}}\}_{s \in S}, \{Cn^{\mathrm{A}}\}_{Cn \in CN}, \{Op^{\mathrm{A}}\}_{Op \in OP}, \{Pr^{\mathrm{A}}\}_{Pr \in PR})$, where for all $s \in S, w = s_1 \ldots s_n \in S^+$

- s^{A} is a set;
- for each constant symbol $Cn \in CN_s$, $Cn^{\mathrm{A}} \in s^{\mathrm{A}}$;
- for each operation symbol $Op \in OP_{w,s}$, $Op^{\mathrm{A}} \in (s_1^{\mathrm{A}} \times \ldots \times s_n^{\mathrm{A}} \to s^{\mathrm{A}})$;
- for each predicate symbol $Pr \in PR_w$
  $Pr^{\mathrm{A}} \in (s_1^A \times \ldots \times s_n^A \to \{TRUE, FALSE\})$ or equivalently $Pr^{\mathrm{A}} \subseteq s_1^A \times \ldots \times s_n^A$;
- if $a, a' \in s^{\mathrm{A}}$, $Ok_s^{\mathrm{A}}(a)$ and $Ok_s^{\mathrm{A}}(a')$, then $a = a'$;
- $Ok_s^{\mathrm{A}}(Cn^{\mathrm{A}})$
- if $Ok_s^{\mathrm{A}}(Op^{\mathrm{A}}(a_1, \ldots, a_n))$, then $Ok_{s_1}^{\mathrm{A}} a_1)$, ..., $Ok_{s_n}^{\mathrm{A}}(a_n)$;
- if $Pr^{\mathrm{A}}((a_1, \ldots, a_n))$, then $Ok_{s_1}^{\mathrm{A}} a_1)$, ..., $Ok_{s_n}^{\mathrm{A}}(a_n)$.

Figure 2: Algebra.

- for each sort s, if a, a' belong to the carrier of s and are not good elements, then a must be equal to a'; i.e. there is at most an element corresponding to error (or if there are several ones they are considered equivalent);
- for each constant symbol Cn: sa $Ok_s(Cn^{\mathrm{A}})$ is true;
- the functions must be "strict", i.e. if the interpretation of an operation symbol applied to some arguments returns a good element, then also the arguments are good elements and analogously for the predicates, if the interpretation of a predicate symbol applied to some arguments is true, then the arguments are good elements:
  - for each operation symbol $Op: s_1 \ldots s_n \to s$, if $Ok_s^{\mathrm{A}}(Op^{\mathrm{A}}(a_1, \ldots, a_n))$ is true, then $Ok_{s_1}^{\mathrm{A}}(a_1)$, ..., $Ok_{s_n}^{\mathrm{A}}(a_n)$ are true;
  - for each predicate symbol $Pr: s_1 \ldots s_n$, if $Pr^{\mathrm{A}}(a_1, \ldots, a_n)$ is true, then $Ok_{s_1}^{\mathrm{A}}(a_1)$, ..., $Ok_{s_n}^{\mathrm{A}}(a_n)$ are true.

Consider the operation symbol `Put: nat stack -> stack` belonging to the signature `S_Stack`, whose interpretation is the function

$PUT\colon \mathbb{N} \cup \{\#nat\} \times \mathbb{N}^* \cup \{\#stack\} \to \mathbb{N}^* \cup \{\#stack\}$;
it is easy to check that if $PUT(n,s)$ is a good element of $\mathbb{N}^* \cup \{\#stack\}$, i.e. if $OK_{stack}(PUT(n,s))$ is true, then also n and s are good elements respectively of $\mathbb{N} \cup \{\#nat\}$ and $\mathbb{N}^* \cup \{\#stack\}$, i.e. $OK_{nat}(n)$ and $OK_{stack}(s)$ are true. Consider now the predicate symbol `Is_Empty: stack` belonging to the same signature, whose interpretation, *IS_EMPTY*, is true only on Λ; also in this case is trivial to see that if *IS_EMPTY* is true on s, then s is a good element, since s may be only Λ. These considerations are examples of the fact that the algebra STACK satisfies the conditions of strictness of the functions and of the predicates. Moreover in STACK the error elements are unique and the constants represent good elements.

1.1.3 Terms and atoms

Now we see how a signature gives the syntax for handling the elements of an algebra (representing elements and elementary conditions on them). For each signature $\Sigma = (S, CN, OP, PR)$ we can consider the set of the expressions, usually called *terms* (that represent elements of the algebras) and *atoms* (that represent elementary conditions on the elements of the algebras) on the signature and on a family of sets of variables X indexed on S; notice that the variables are needed for representing generic elements of the algebras. Such expressions are inductively defined as follows:

i) for each variable $x \in X$ of sort s, x is a term of sort s;

ii) for each constant $Cn\colon s \in CN$, Cn is a term of sort s;

iii) for each operation symbol $Op\colon s_1 \dots\ s_n \to s \in OP$, if $t_1, \dots, t_n$ are terms respectively of sorts $s_1, \dots, s_n$, then $Op(t_1, \dots, t_n)$ is a term of sort s;

iv) for each predicate symbol $Pr\colon s_1 \dots s_n \in PR$, if $t_1, \dots, t_n$ are terms respectively of sorts $s_1, \dots, s_n$ then $Pr(t_1, \dots, t_n)$ is an atom;

v) for each $s \in S$, if t_1, t_2 are both terms of sort s, then $t_1 = t_2$ is an atom.

The intuitive meaning is that each term of sort s given a value to the variables appearing in it (we require that to the variables are always assigned good values) is a syntactic representation of an element of the carrier associated with s; for example the term `Get(Put(0,Empty))` represents the element Λ of the carrier associated with the sort `stack`; while each atom represents the truth of elementary conditions (atomic) on the elements of the various carriers; for example the atom `Is_Empty(Empty)` represents the truth of the condition "the stack represented by `Empty` is empty".

Is important to note that a term may represent also the error element; for example the term `First(Empty)`, the first element of the empty stack, represents $\#nat$, the error of sort `nat`.

It should be clear that in general an element in a particular algebra may be represented by different terms; for example the terms `Get(Put(0,Empty))` and `Empty` represent the same stack Λ.

Consider the atom with variables `Is_Empty(Put(x,Empty))`; it represents the truth of the condition "for each values assigned to the variable `x` the stack represented by

Put(x,Empty) is empty", that is false whatever is the value assigned to x, since the term Put(x,Empty) represents a stack having exactly one element.

The equality among terms; it is a kind of implicit predicate, $\_ = \_ : s\ s$ for each $s \in S$; thus if t_1, t_2 are both terms of sort s, then $t_1 = t_2$ is an atom that, for an appropriate assignment of values to the variables, represents the truth of the condition "t_1, t_2 represent the same element".

The atom with variables First(Put(x,Empty)) = y, assigned the values to the two variables x and y, say 1 and 3, represents the condition "does First(Put(1,Empty)) represent 3?", that is false since First(Put(1,Empty)) represents the natural 1. If instead the values of x and y are respectively 2 and 2, then the atom First(Put(x,Empty) = y represents the condition "does First(Put(2,Empty)) represent 2?" that is true.

We know that since there is a unique error element, i.e. we have that assigned the values to the variables appearing in the two terms, if t_1 represents the error and if $t_1 = t_2$ is true, then also t_2 represents the error.

The equality Get(Put(0,Empty)) = Empty that is true since both the terms represent the same stack Λ; notice that it is sufficient to know that one of the two terms is not error, for example Empty which is a constant of sort stack, for concluding that also the other term, Get(Put(0,Empty)), is not error.

Consider now the equality First(Get(Put(0,Empty))) = First(Empty) that is true and First(Empty) represents the error on the stacks, since we cannot get the first element from an empty stack, then also First(Get(Put(0,Empty))) represents the error on the stacks.

The element of an algebra A represented by a term t, after having assigned values to the variables by means of a variable evaluation V, is said *interpretation of t in* A *w.r.t. V* and is written $t^{A,V}$; analogously the truth value represented by an atom α, after having assigned some values to its variables by means of a variable evaluation V, is said *interpretation of α in* A *w.r.t. V* and is written $\alpha^{A,V}$.

The formal definition of terms and atoms on a signature and a family of sets of variables and their interpretations is reported in Fig. 3.

Example 1.1 Concrete data types finite sets and finite multisets of naturals
The signature S_Set for the data type finite sets of naturals is:

```
sortnat set
cn  Zero:  nat
cn  Empty:  set
op  Succ:  nat -> nat
op  Singleton:  nat -> set
op  Union:  set set -> set
pr  Is_In:  nat set
pr  OKnat:  nat
pr  OKset:  set
```

The data type finite set of naturals is represented by the S_Set-algebra SET defined by:

- Carriers

Let $\Sigma = (S, CN, OP, PR)$ be a signature, $X = \{X_s\}_{s \in S}$ a family indexed on S of sets of variables and A a Σ-algebra.

- $\{T_\Sigma(X)_s\}_{s \in S}$ the family of the sets of *terms* on Σ and X is inductively defined as follows; for each $s \in S$:
  - $X_s \subseteq T_\Sigma(X)_s$;
  - $CN_s \subseteq T_\Sigma(X)_s$;
  - for each $Op\colon s_1 \dots s_n \to s \in OP$, $t_1 \in T_\Sigma(X)_{s_1}, \dots, t_n \in T_\Sigma(X)_{s_n}$, $Op(t_1, \dots, t_n) \in T_\Sigma(X)_s$.
- $A_\Sigma(X)$ the set of the *atoms* on Σ and X is inductively defined as follows:
  - for each $Pr\colon s_1 \dots s_n \in PR$, $t_1 \in T_\Sigma(X)_{s_1}, \dots, t_n \in T_\Sigma(X)_{s_n}$, $Pr(t_1, \dots, t_n) \in A_\Sigma(X)$;
  - for each $s \in S$, $t_1, t_2 \in T_\Sigma(X)_s$, $t_1 = t_2 \in A_\Sigma(X)$.
- Given an evaluation V of the variables of X in A (i.e. a function from X into A respecting the sorts and s.t. $Ok_s^{\mathrm{A}}(V(x))$ for each $x \in X_s$), the *interpretation* of the terms of $T_\Sigma(X)$ and of the atoms of $A_\Sigma(X)$ in A w.r.t. V is inductively defined as follows, (if $t \in T_\Sigma(X)$ and $\alpha \in A_\Sigma(X)$, the interpretations in A w.r.t. V of t and α are denoted respectively by $t^{\mathrm{A},V}$ and $\alpha^{\mathrm{A},V}$):
  - $x^{\mathrm{A},V} = V(x)$;
  - $Cn^{\mathrm{A},V} = Cn^{\mathrm{A}}$;
  - $Op(t_1, \dots, t_n)^{\mathrm{A},V} = Op^{\mathrm{A}}(t_1^{\mathrm{A},V}, \dots, t_n^{\mathrm{A},V})$;
  - $Pr(t_1, \dots, t_n)^{\mathrm{A},V} = \mathit{TRUE}$ if $(t_1^{\mathrm{A},V}, \dots, t_n^{\mathrm{A},V}) \in Pr^{\mathrm{A}}$, FALSE otherwise;
  - $(t_1 = t_2)^{\mathrm{A},V} = \mathit{TRUE}$ if $t_1^{\mathrm{A},V} = t_2^{\mathrm{A},V}$, FALSE otherwise.

Figure 3: Terms and atoms on a signature and their interpretation.

  - $\texttt{nat}^{\mathrm{SET}} = \mathbb{N} \cup \{\#nat\}$
  - $\texttt{set}^{\mathrm{SET}} = \mathcal{S}(\mathbb{N}) \cup \{\#set\}$
- Interpretation of constant symbols
  - $\texttt{Zero}^{\mathrm{SET}} = 0$
  - $\texttt{Empty}^{\mathrm{SET}} = \emptyset$
- Interpretation of operation symbols
  - $\texttt{Succ}^{\mathrm{SET}}\colon \texttt{nat}^{\mathrm{SET}} \to \texttt{nat}^{\mathrm{SET}}$
    $\texttt{Succ}^{\mathrm{SET}}(n) = \#nat$ if $n = \#nat$, $n + 1$ otherwise

  - $\texttt{Singleton}^{\mathrm{SET}}: \texttt{nat}^{\mathrm{SET}} \to \texttt{set}^{\mathrm{SET}}$
    $\texttt{Singleton}^{\mathrm{SET}}(n) = \#set$ if $n = \#nat$, $\{n\}$ otherwise
  - $\texttt{Union}^{\mathrm{SET}}: \texttt{set}^{\mathrm{SET}} \times \texttt{set}^{\mathrm{SET}} \to \texttt{set}^{\mathrm{SET}}$
    $\texttt{Union}^{\mathrm{SET}}(s_1, s_2) = \#set$ if either $s_1 = \#set$ or $s_2 = \#set$, $s_1 \cup s_2$ otherwise

- Interpretation of predicate symbols
  - $\texttt{Is\_In}^{\mathrm{SET}}: \texttt{nat}^{\mathrm{SET}} \times \texttt{set}^{\mathrm{SET}} \to \{\mathit{TRUE}, \mathit{FALSE}\}$
    $\texttt{Is\_In}^{\mathrm{SET}}(n, s) = \mathit{TRUE}$ if $n \in s$, $n \neq \#nat$ and $s \neq \#set$, FALSE otherwise
  - $\texttt{OKnat}^{\mathrm{SET}}: \texttt{nat}^{\mathrm{SET}} \to \{\mathit{TRUE}, \mathit{FALSE}\}$
    $\texttt{OKnat}^{\mathrm{SET}}(n) = \mathit{TRUE}$ if $n \neq \#nat$, FALSE otherwise
  - $OKset^{\mathrm{SET}}: \texttt{set}^{\mathrm{SET}} \to \{\mathit{TRUE}, \mathit{FALSE}\}$
    $OKset^{\mathrm{SET}}(s) = \mathit{TRUE}$ if $s \neq \#set$, FALSE otherwise

The algebra SET thus represents the data type finite sets of naturals. But the same signature may be used also for another data type: the finite multisets of naturals, as the following `S_Set` algebra MSET shows.

- Carriers
  - $\texttt{nat}^{\mathrm{MSET}} = \mathbb{N} \cup \{\#nat\}$
  - $\texttt{set}^{\mathrm{MSET}} = M(\mathbb{N}) \cup \{\#set\}$,
    where $M(\mathbb{N})$ denotes the set of all the finite multisets of naturals[1], i.e., finite sets with possible multiple occurrences of the same element

- Interpretation of constant symbols
  - $\texttt{Zero}^{\mathrm{MSET}} = 0$
  - $\texttt{Empty}^{\mathrm{MSET}} = \emptyset$, where $\emptyset: \mathbb{N} \to \mathbb{N}$ is the function s.t. $\emptyset(n) = 0$ for each n

- Interpretation of operation symbols
  - $\texttt{Succ}^{\mathrm{MSET}}: \texttt{nat}^{\mathrm{MSET}} \to \texttt{nat}^{\mathrm{MSET}}$
    $\texttt{Succ}^{\mathrm{MSET}}(n) = \#nat$ if $n = \#nat$, $n + 1$ otherwise
  - $\texttt{Singleton}^{\mathrm{MSET}}: \texttt{nat}^{\mathrm{MSET}} \to \texttt{set}^{\mathrm{MSET}}$
    $\texttt{Singleton}^{\mathrm{MSET}}(n) = \#set$ if $n = \#nat$, $f: \mathbb{N} \to \mathbb{N}$ otherwise,
    where $f(n) = 1$ and $f(h) = 0$ for each $h \neq n$
  - $\texttt{Union}^{\mathrm{MSET}}: \texttt{set}^{\mathrm{MSET}} \times \texttt{set}^{\mathrm{MSET}} \to \texttt{set}^{\mathrm{MSET}}$
    $\texttt{Union}^{\mathrm{MSET}}(s_1, s_2) = \#set$ if either $s_1 = \#set$ or $s_2 = \#set$,
    $s_1 + s_2: \mathbb{N} \to \mathbb{N}$ otherwise
    where $(s_1 + s_2)(n) = s_1(n) + s_2(n)$ for each n

- Interpretation of predicate symbols

[1] See the mathematical notations at the beginning of the manual.

- $\texttt{Is\_In}^{\text{MSET}}$: $\texttt{nat}^{\text{MSET}} \times \texttt{set}^{\text{MSET}} \to \{\mathit{TRUE}, \mathit{FALSE}\}$
  $\texttt{Is\_In}^{\text{MSET}}(n, s) = \mathit{TRUE}$ if $s(n) \neq 0$, $n \neq \#nat$ and $s \neq \#set$,
  FALSE otherwise
- $\texttt{OKnat}^{\text{MSET}}$: $\texttt{nat}^{\text{MSET}} \to \{\mathit{TRUE}, \mathit{FALSE}\}$
  $\texttt{OKnat}^{\text{MSET}}(n) = \mathit{TRUE}$ if $n \neq \#nat$, FALSE otherwise
- $OKset^{\text{MSET}}$: $\texttt{set}^{\text{MSET}} \to \{\mathit{TRUE}, \mathit{FALSE}\}$
  $OKset^{\text{MSET}}(s) = \mathit{TRUE}$ if $s \neq \#set$, FALSE otherwise

End example

1.2 Abstract data types

The definition of the data type stack given in Sect. 1.1, the algebra STACK, corresponds to the our intuition but it is concrete in the sense that we have chosen a particular model of the stacks. In the algebra STACK the stacks are realized concretely as streams, but we can also imagine another algebra STACK$'$ defined analogously to STACK, where the stacks are sets of pairs (n, s), s position of n in the stacks; also the carrier of the sort `nat` may be the naturals as either binary, decimal or hexadecimal streams.

Now we want to give an *abstract definition* of the same data type so that several different models may be all considered as concrete realizations of such abstract structure. To do that we identify the algebras that have exactly the same structure, i.e. which are isomorphic, and then we choose one as their representative. (Informally, two algebras are *isomorphic* iff for each sort there exists a bijective correspondence between the carriers of such sort in the two algebras, and the constants, operations and the truth of the predicates preserve this correspondence). Thus with *abstract data type* we mean a class algebras on the same signature closed by isomorphism (i.e. including each algebra isomorphich to one of its elements).

Defining an abstract data type directly, i.e. as class of isomorphic algebras, is not very convenient. Indeed it is sufficient to consider the simple case of the sets and multisets (see Ex. 1.1) for imagining what means to repeat this procedure for more complex structures. The idea is to specify the abstract data type by giving:

- the syntax for representing/handling its elements as a signature and
- the properties that the interpretations of the syntactic symbols (symbols of constants, of operations and of predicates) should satisfy.

For example we may want to express that:

- the addition between naturals is commutative,
- there exists the neutral element w.r.t. addition,
- the term `Succ(n)` represents a positive natural for all `n`,
- the stack with an element is not empty,
- the addition of a positive natural with each other natural is still a positive natural,

– if two naturals are one bigger than or equal of the other and vice versa, then they are equal,

– and so long.

As it is possible to express such properties in a precise way? We have to use:

- equalities between terms for expressing that some terms represent the same element;

  addition is commutative: $n_1 + n_2 = n_2 + n_1$, with n_1 and n_2 of sort nat;

  0 is the neutral element of addition: $n + 0 = n$, with n of sort nat;

- truth of predicates for expressing that some conditions hold on the elements of the carriers;

  each successor of a natural is positive: $Succ(n) > 0$, with n of sort n;

  the stack with at least an element is not empty:

  $\neg\ Is\_Empty(Put(n, s))$, with n of sort nat and s of sort $stack$;

- relationships among equalities of terms and/or truth of atoms;

  if a natural is positive, adding it to another natural still gives a positive natural:

  $n > 0 \ \supset\ n + m > 0$, with n, m of sort nat;

  if a natural is bigger or equal to another one and vice versa, then they are equal:

  $n \geq m\ \wedge\ m \geq n\ \supset\ n = m$, with n, m of sort nat;

  if a natural n is either equal or bigger of another one m, then $n4$ is bigger or equal to m:

  $n = m\ \vee\ n > m\ \supset\ n \geq M$, with n, m of sort nat;

  for each natural there exists a bigger one: $\forall\, n\ .\ (\exists\, m\ .\ m > n)$.

The properties, as the ones listed above, are formalized by axioms, i.e. logic formulae, for which the first-order logic is appropriate.

In Fig. 4 is reported the precise definition of first-order formulae.

Now we have to make precise when a concrete data type represented by an algebra satisfies a property expressed by a formula, i.e. when a formula holds on an algebra. In Fig. 5 is reported the precise definition of validity in an algebra of the first-order formulae.

An algebra A is a *model* of a formula ϕ if ϕ *holds* in A, *model of a set of formulae* if A is model of each formula belonging to the set.

Example 1.2 Evaluation of first-order formulae
Let STACK be the algebra given in Sect. 1.1 representing the data type of the stacks. We show the evaluation of some formulae.

– STACK, V $\models$`Is_Empty(Empty)`

  since for each V $\texttt{Is\_Empty(Empty)}^{\mathrm{STACK},V} = \texttt{Is\_Empty}^{\mathrm{STACK}}(\texttt{Empty}^{\mathrm{STACK},V}) = IS\_EMPTY(\Lambda) = TRUE$.

Let $\Sigma = (S, CN, OP, PR)$ be a signature, X a family of sets of variables indexed on S. The set $F_\Sigma(X)$ of the *first-order logic formulae on* Σ *and* X is inductively defined as follows.

- $A_\Sigma(X) \subseteq F_\Sigma(X)$ ($A_\Sigma(X)$, the set of the atoms, is defined in Fig. 2)
- $\phi_1 \wedge \phi_2 \in F_\Sigma(X)$ for each $\phi_1, \phi_2 \in F_\Sigma(X)$ (conjunction)
- $\phi_1 \vee \phi_2 \in F_\Sigma(X)$ for each $\phi_1, \phi_2 \in F_\Sigma(X)$ (disjunction)
- $\phi_1 \supset \phi_2 \in F_\Sigma(X)$ for each $\phi_1, \phi_2 \in F_\Sigma(X)$ (implication)
- $\neg \phi \in F_\Sigma(X)$ for each $\phi \in F_\Sigma(X)$ (negation)
- $\forall x . \phi$ for each $\phi \in F_\Sigma(X)$ and $x \in X$ (universal quantification)
- $\exists x . \phi \in F_\Sigma(X)$ for each $\phi \in F_\Sigma(X)$ and $x \in X$ (existential quantification)

Figure 4: Formulae of the first-order logic.

- STACK, $V \models$`First(Put(x,Empty)) = y` with V s.t. $V(\mathtt{x}) = 2$ and $V(\mathtt{y}) = 2$ since `First(Put(x,Empty))`$^{\mathrm{STACK},V} = \mathit{FIRST}(\mathit{PUT}(V(\mathtt{x}), \Lambda)) = \mathit{FIRST}(\mathit{PUT}(2, \Lambda)) = 2$ and $\mathtt{y}^{\mathrm{STACK},V} = 2$
- STACK, $V' \not\models$`First(Put(x,Empty)) = y` with V' s.t. $V'(\mathtt{x}) = 1$ and $V'(\mathtt{y}) = 2$ since otherwise, analogously to the previous case, we would get $1 = 2$.

End example

An abstract data type is specified by giving a signature and a set AX of axioms (first-order formulae) representing the properties of the interpretations of the constant, operation and predicate symbols. The pair signature and set of axioms is said *algebraic specification*.

An algebraic specification (Σ, AX) determines a set of concrete data types (Σ-algebras), all that which are models of the set of axioms AX; such algebras are said *models of the specification*.

Notice that in the models of AX also formulae not belonging to AX hold; indeed the property of the algebras and the definition of validity of the formulae are such that the equalities between terms and the truth of the atoms that hold in the models of the specification are not only that explicitly expressed by the axioms, but also other implicitly implied by them, i.e. that *logically follow from the axioms*. In other words it is as if the set of the axioms AX includes also all that formulae which are logical consequences of them. This allow to express the properties of the abstract data type represented by the specification in a syntectic way without any redundancy, with a set of formulae which is not too much complex and more readable.

Let Σ be a signature, X a family of sets of variables indexed on S, $\phi \in F_\Sigma(X)$ and A a Σ-algebra.
We say that ϕ *holds* in A (and write $A \models \phi$) iff for each variable evaluation $V: X \to A$ A satisfies ϕ w.r.t. V (we write $A, V \models \phi$). The inductive definition of satisfiability is as follows.

- $A, V \models \alpha$ iff $\alpha^{A,V}$ is true
- $A, V \models \phi_1 \wedge \phi_2$ iff $A, V \models \phi_1$ and $A, V \models \phi_2$
- $A, V \models \phi_1 \vee \phi_2$ iff either $A, V \models \phi_1$ or $A, V \models \phi_2$ (also both)
- $A, V \models \phi_1 \supset \phi_2$ iff either $A, V \not\models \phi_1$ or $A, V \models \phi_2$
- $A, V \models \neg \phi$ iff $A, V \not\models \phi$
- $A, V \models \forall x . \phi$ iff for each $v \in A_s$, with s sort of x, $A, V[v/x] \models \phi$
- $A, V \models \exists x . \phi$ iff there exists $v \in A_s$, with s sort of x, s.t. $A, V[v/x] \models \phi$

where $V[v/x]$ is the evaluation s.t. $V[v/x](x) = v$ and for each $y \neq x$ $V[v/x](y) = V(y)$

Figure 5: Validity of the first-order formulae.

Given an algebraic specification, for each algebra A model of such specification, we have that:

- the equality symbol is intended in strong sense, i.e. if $t_1 = t_2$ holds in A for a variable evaluation V and the interpretation of t_1 is a good element of sort s ($Ok_s^A(t_1^{A,V})$ is true), then also the interpretation of t_2 is a good element ($Ok_s^A(t_2^{A,V})$ is true) and vice versa, as already seen (since there is a unique error element). It is as if among the axioms of the specification there are the formulae
$$x = y \wedge Ok_s(x) \supset Ok_s(y) \quad \text{and} \quad x = y \wedge Ok_s(y) \supset Ok_s(x);$$
- the equality symbol is reflexive, i.e. $t = t$ holds in A for each variable evaluation; it is as if among the axioms of the specification there is the formula $x = x$;
- the equality symbol is symmetric, i.e. if $t_1 = t_2$ holds in A for a variable evaluation V, then $t_2 = t_1$ holds in A for V; it is as if among the axioms of the specification there is the formula $x = y \supset y = x$;
- the equality symbol is transitive, i.e. if $t_1 = t_2$ and $t_2 = t_3$ hold in A for a variable evaluation V, then $t_1 = t_3$ holds in A for V; it is as if among the axioms of the specification there is the formula $x = y \wedge y = z \supset x = z$;

- the equality symbol respects the operations and the predicates, i.e. if $t_i = t'_i$, for $i = 1, \ldots, n$ hold in A for a variable evaluation V, then $Op(t_1, \ldots, t_n) = Op(t'_1, \ldots, t'_n)$ and $Pr(t_1, \ldots, t_n) \supset Pr(t'_1, \ldots, t'_n)$ hold in A for V, with $Op: s_1 \ldots s_n \to s$ and $Pr: s_1 \ldots s_n$. It is as if among the axioms of the specification there are the formulae

  $x_1 = x'_1 \ \wedge \ \ldots \ \wedge \ x_n = x'_n \ \supset \ Op(x_1, \ldots, x_n) = Op(x'_1, \ldots, x'_n)$,

  $x_1 = x'_1 \ \wedge \ \ldots \ \wedge \ x_n = x'_n \ \supset \ (Pr(x_1, \ldots, x_n) \ \supset \ Pr(x'_1, \ldots, x'_n))$;

- always good values are assigned to the variables, i.e. for all variable evaluation V, $Ok_s^{\mathrm{A}}(V(x))$, with x variable of sort s; this allows to avoid to explicitly writing in the set of axioms that a variable is defined. It is as if each axiom of the specification is prefixed by $Ok_{s_1}(x_1) \ \wedge \ \ldots \ \wedge \ Ok_{s_n}(x_n) \ \supset$, for all variables $x_1, \ldots, s_n$ appearing in the axiom;

- the constants represent always good values, i.e. $Ok_s(Cn)$ holds in A w.r.t. V with $Cn: s$ it is as among the axioms of the specification there are the formulas $Ok_s(Cn)$, for all constants Cn;

- the functions and the predicates are strict as already required in the definition of algebra given in Fig. 1; i.e. if a symbol of either operation or of predicate applied to some arguments is defined, i.e. represents a good element, then also the arguments are good elements. It is as if among the axioms of the specification there are the formulae, for all Op and Pr

  $Ok_s(Op(x_1, \ldots, x_n)) \ \supset \ Ok_{s_i}(x_i)$ with $i = 1, \ldots, n$ and

  $Pr(x_1, \ldots, x_n) \ \supset \ Ok_{s_i}(x_i)$ with $i = 1, \ldots, n$.

Example 1.3 Properties of the data type stack
The axioms characterizing the stacks are:

```
Get(Put(y,x)) = x    First(Put(y,x)) = y
OKnat(Succ(x))    not Is_Empty(Puty,x))    Is_Empty(Empty)
```

We see that in this case the above formulae are sufficient for expressing all the required properties. For example, we show how to verify which are the good elements of the carrier of the sort `stack`, using the only explicit property given by the formula `OKstack(Empty)`. We use the properties of strictness of the functions and of the predicates, the fact that equality is strong, the definedness of the variables and also the formulae in the set of axioms. The proof is by induction:

- The implicit axiom `OKstack(Empty)`, asserts that `Empty` represents a good element, belonging to the carrier of the sort `stack`;

- assuming that `x` represents a stack of $n-1$ good elements (`OKstack(x)`) and that `y` represents a good element (`OKnat(y)`), we show that the stack of n elements represented by `Put(y,x)` is a good element of the carrier of the sort `stack`, using the axiom `Get(Put(y,x)) = x`; indeed, for the strong equality and the definedness of the variables, by `OKstack(x)` we have that `Get(Put(y,x))` represents a good stack, i.e. `OKstack(Get(Put(y,x)))` logically follows from the axioms; for the strictness of the functions by `OKstack(Get(Put(y,x)))` we get that `Put(y,x))` represents a good stack, i.e. that `OKstack(Put(y,x)))` logically follows from the axioms.

End example

Given an algebraic specification (Σ, AX), we have seen which are its models, i.e. the algebras on the signature Σ satisfying the axioms AX. Now we have the problem: which abstract data type is determined by such specification? As first thing we have to recall that an abstract data type is an isomorphism class of models and when we speak of a particular model we intend a representative of the class to whom it belongs.

Initial approach The initial approach chooses among the models of an algebraic specification a particular model, said the *initial model*, as *the abstract data type defined by the algebraic specification*. An initial model is characterized by the following properties:

1. each element in the algebra is represented by a closed term (i.e. in which no variable appear) on the signature; this means that all the elements are representable, i.e. each element may be represented syntactically by a term.

2. only the identifications between elements forced by the axioms hold (i.e. the interpretations of two terms are equal if and only if the equality between them logically follows from the axioms);

3. an atom is true iff it is forced to be true by the axioms (i.e. iff its truth logically follows from the axioms).

It may be useful to synthetise the above properties with the slogan:

"only what it is said by the axioms holds".

We esemplify the property 1) with reference to the example of the algebra STACK on the signature `S_Stack`: each element of the algebra belonging to the carrier $\mathbb{N}^* \cup \{\#stack\}$ is represented syntactically by a term without variables of sort `stack`. Consider the simplest form that a closed term of sort `stack` which represents a stack of x naturals $(x \geq 0)$ may have:

$\mathtt{Put(n_{x-1},Put(n_{x-2},\ldots\ Put(n_0,Empty)\ \ldots))}$,

with $\mathtt{n}_{x-1}$, $\mathtt{n}_{x-2}$, $\ldots$, $\mathtt{n}_0$ terms without variables of sort `nat` representing x natural numbers; this is an example of term without variables of sort `stack` which represents syntactically the stack with such x natural numbers. But recall that such term is not the unique possible, since several closed terms on the signature may have the same interpretation in the algebra, i.e. they may be the syntactic representation of the same element of the algebra, for example `Get(Put(Zero,Empty))` and `Empty` represent both the empty stack.

Notice that, given a specification, the conditions 1), 2), 3) make automatically false in the initial model everything that is not logic consequence of the axioms; indeed since all elements are representable, all identifications between terms logically follow from the axioms and the truth of all conditions on the elements logically follow from the axioms. In conclusion, only the formulae that logically follow from the axioms hold; thus during the development of an initial specification we have to formalize all and only the properties that we want to hold in the data type.

It may happen that the initial model does not exist, as it is shown in the following example.

Example 1.4

```
sorts
cn  A, B, C: s
ax  A = B or A = C
```

The possible models of the above specification for which 1) holds, i.e. the algebras on its signature in which the axiom holds and where all the elements are representable, are isomorphic to one of the following:

- the algebra M whose only carrier is $\{1\}$ and the interpretation of the constants is $\mathtt{A}^{\mathrm{M}} = \mathtt{B}^{\mathrm{M}} = \mathtt{C}^{\mathrm{M}} = 1$, i.e. the algebra in which all the constants are identified;
- the algebra M′ whose only carrier is $\{1, 2\}$ and the interpretation of the constants is $\mathtt{A}^{\mathrm{M}'} = \mathtt{B}^{\mathrm{M}'} = 1$, $\mathtt{C}^{\mathrm{M}'} = 2$, i.e. the algebra in which the constants `A` and `B` are identified and
- the algebra M″ whose only carrier is $\{1, 2\}$ and the interpretation of the constants is $\mathtt{A}^{\mathrm{M}''} = \mathtt{C}^{\mathrm{M}''} = 1$, $\mathtt{B}^{\mathrm{M}''} = 2$, i.e. the algebra in which the constants `A` and `C` are identified.

These algebras are the unique, modulo isomorphism, satisfying the axiom, i.e. which are models of the specification; indeed the axiom requires that there are at most two elements in the carrier, i.e. that the three constants represent either a unique element or at most two.

The initial model does not exist, indeed there is not a unique (modulo isomorphism) model in which the number of identifications among the terms is minimum, since we have two models M′ and M″ with the same number of identifications which are not isomorphic.
End example

A way for ensuring the existence of the initial model is to use axioms of particular form, precisely positive conditional formulae in the first-order logic[2], i.e. formulae having form:

$$\phi_1 \ \wedge \ \ldots \ \wedge \ \phi_n \ \supset \ \phi,$$

where $n \geq 0$, for i s.t. $1 \leq i \leq n$ ϕ_i has either form $Ok(t) \ \wedge \ Ok(t') \ \wedge \ t = t'$ or it is an atom built by a predicate symbol and ϕ is either an atom or the negation of an atom.

Note that when $n = 0$ the premises of such axioms are always satisfied, thus in these cases we simply write ϕ.

Example 1.5 Positive conditional formulae and not

- If a natural n belongs to the sets s_1 and s_2, then it belongs to their union, i.e.:

  $n \in s_1 \ \wedge \ n \in s_2 \ \supset \ n \in s_1 \cup s_2$;

  this formula is positive conditional since each ϕ_i and also ϕ are atoms built by the predicate symbol "$\in$".

[2]Notice that the positive conditional formulae in the first-order logic are just the Horn clauses used by the logic programming language Prolog.

– If a natural n belongs at least to a set between s_1 and s_2, then it belongs to their union, i.e.:

  $n \in s_1 \ \vee \ n \in s_2 \ \supset \ n \in s_1 \cup s_2$;

  this formula is not positive conditional, indeed it is not of the right form since each ϕ_i may be not a disjunction between formulae, but only either an atom or a formula of the form $Ok(t) \ \wedge \ Ok(t') \ \wedge \ t = t'$.

End example

In general it is convenient to define the specifications of data types in a modular way using appropriate constructs for structuring the specifications. In this way it is possible to reuse specifications appearing in several parts of the global specification and to get a better clarity and understandability of the structure of the defined data type. The common constructs for structuring the specifications are reported in the user manual [?].

Loose approach In the loose approach an algebraic specification determines a class of abstract data types: all that satisfying *at least* all the axioms of the specification. In this case a specification expresses the general properties of a data type, and thus in general the axioms are not conditional formulae but more complex formulae of the first-order logic.

The loose approach is different from the initial one since, in the initial approach we require that in the abstract data type determined by the specification:

– the elements are representable and

– the truth of atoms and of equalities between terms should logically follow from the axioms;

this implies that:

> everything which does not logically follow from the axioms is false in the abstract data type determined by the specification.

To the contrary, in the loose approach the abstract data types determined by the specification should not satisfy particular requirements, this implies that:

> the interpretation of all formulae that do not logically follow from the axioms is free, i.e. one abstract data type may satisfy the formula and another no.

In conclusion the point of view of the loose approach is different, since we have no conditions on all that it is neither explicitly expressed by the axioms, nor logically follow from them, to the contrary the explicit formulae of the specification, i.e. the axioms, and those which logically follow are used for expressing the properties that we want that are surely satisfied.

Example 1.6 The abstract data type semigroup
We want to represent the abstract data type semigroup characterized by having a binary and associative operation on the elements of the data type, i.e. an operation that taken

two elements returns an element of the data type and such that the result of several applications of the operation to several arguments is independent by the order in which they are performed. Since the required properties are very general and we are not interested into the structure of the elements of the data type the adequate approach in this case is the loose one.

Consider the following specification:

```
SEMIGROUP =
requirement
    sort   s
    op  _ @ _ :  s s -> s
    ax  (x @ y) @z = x @(y @z)
end
```

With the loose approach the specification represents the class of abstract data types (also non isomorphic between them), that satisfy at least the property required by the axiom.

Such specification requires that the algebra has one carrier and that there is an associative operation on the elements of the carrier which taken two elements returns a element of such carrier. In conclusion each algebra with a carrier and a binary associative operation on the elements of such carrier, i.e. satisfying the axiom, is, modulo isomorphism, an abstract data type belonging to the class, whatever are the other properties satisfied by the algebra; thus this specification represents the class of all semigroups.

Now we show two algebras which are, modulo isomorphism, two abstract data types belonging to the class; the first algebra A is defined by:

Carrier $\mathtt{s}^{\mathrm{A}} = \mathbb{Z}$

Interpretation of the operation symbol

$\_@^{\mathrm{A}}\_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \qquad x@^{\mathrm{A}}y = x + y;$

The second algebra B is defined by:

Carrier $\mathtt{s}^{\mathrm{B}} = \{0, 1\}$

Interpretation of the operation symbol

$\_@^{\mathrm{B}}\_ : \{0,1\} \times \{0,1\} \to \{0,1\}$

$0@^{\mathrm{B}}0 = 0, \qquad 0@^{\mathrm{B}}1 = 1, \qquad 1@^{\mathrm{B}}0 = 1, \qquad 1@^{\mathrm{B}}1 = 0.$

The algebras A, B are, modulo isomorphism, two abstract data types of the class, but they are not the same abstract data type since they are not isomorphic.

Notice that if we follow the initial approach for determining the abstract data type represented by the above specification, we find that it is the empty algebra. Indeed the carrier associated with the sort `s` is the empty set, since each element of the carrier must be representable by a term, but the set of the closed terms, i.e. without variables, on the signature is empty, since there are neither constant terms nor terms built on them.
End example

1.3 Specifications of data types at different levels of abstraction

As we have seen in Sect. 1.2 the two approaches to the specification, initial and loose, differ, since the first, determines only one abstract data type, in which only the properties given by the axioms and those that logically follow from them hold, while the second determines different abstract data types, in which the properties given by the axioms and those which logically follow from them hold and the interpretation of each other logic formula is either true or false depending on the particular data type.

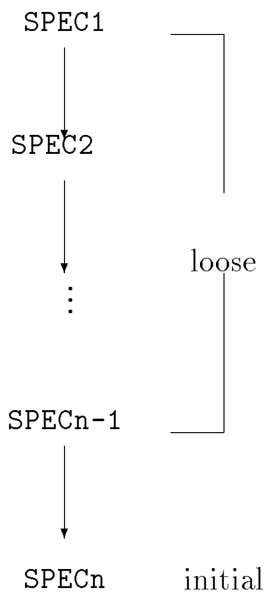


Figure 6: Top-down development of a specification.

What said before suggests that the two approaches may be used for specifying a data type at different levels of abstraction:

- using the loose approach, initially we specify only general properties, without investigating the structure of the data type; then we require that further properties are satisfied by the data type, by adding further requirements that fix the interpretation of an increasing number of formulae and/or specify the structure of the data type in a more detailed way;

- using the initial approach, when we have specified all the properties which have to be satisfied by the abstract date type, we are sure that everything which does not logically follows from the axioms is false in the models of the specification, i.e. we determine only one abstract data type and not a class of abstract data types.

This development of the specification of an abstract data type at different levels of abstraction, made by different steps of refining, gives a sequence of loose specifications, say `SPEC1`, ..., `SPECn-1` and an initial specification `SPECn` representing the data type (see Fig. 6). Clearly the specification given at the i-th step, `SPECi`, must refine, i.e. extend by adding new requirements, to that given to the previous step; i.e. `SPECi` it must implement `SPECi-1`. Formally, `SPECi` *implements* `SPECi-1` whenever all models of α(`SPECi`) are also models of `SPECi-1`, where α is a function between specifications which express how to realize the parts of `SPECi-1` with the parts of `SPECi` and how to get everything that has been added in `SPECi`; in particular α establishes the correspondence between the symbols of the components of the two specifications (how the symbols of `SPECi-1` are changed in its implementation `SPECi`).

Example 1.7 Top-down development of the specification of integers
We specify the integers in a top-down way, by adding step after step new requirements, until we get a specification with initial approach representing the integers with the addition operation.

Consider the specification of Ex. 1.6, with loose approach, which represents the class of the data types semigroup, and call it `INT1`.

We define a specification `INT2`, always with loose approach, which implements `INT1`, but restricts the set of the models since we want to determine the algebras in which the interpretation of the binary operation has a neutral element, i.e. an element such that the function applied to it and to whichever other element returns as result the other element.

Thus `INT2` is defined enriching `INT1` with a constant "`N`" representing the neutral element and with the formulae `x @ N = x` and `N @ x = x`, describing its properties.

```
INT2 =
requirement
    sort    s
    var x y z:  s
    op  _ @ _ :  s s -> s
    ax  (x @ y) @ z = x @(y @ z)
    cn  N: s
    ax  x @ N = x
    ax  N @ x = x
end
```

It is possible to extend the algebras A and B given in Ex. 1.6 to two models of `INT2`, respectively A' and B'.

Consider indeed the algebra A' with the same carrier and the same interpretation of the operation symbol of A and with the interpretation of the constant symbol $\mathtt{N}^{A'} = 0$.

Analogously the algebra B' has the same carrier and the same interpretation of the operation symbol of B and the interpretation of the constant symbol: $\mathtt{N}^{B'} = 0$.

The algebras A', B' are, modulo isomorphism, two abstract data types of the class represented by the specification `INT2`, but they are not the same abstract data type since they are not isomorphic.

Notice that in this case the function among specifications α_1, used for showing that `INT2` implements `INT1`, just hides the constant "`N`".

We further refine `INT2` since we want to determine only the algebras in which the interpretation of the symbol of binary operation has the inverse, i.e. for each element of the carrier we want that there exists an element such that the function applied to them gives as result the neutral element. In conclusion we want to represent the class of the data types groups (semigroup with neutral element and inverse).

Thus we define the specification `INT3`, always with loose approach, that implements `INT2`, obtained by adding an operation symbol, `Inv`, whose interpretation is a function which taken an element returns its inverse, whose properties are described by the formulae `x @ Inv(x) = N` and `Inv(x) @ x = N`. Thus we get the following specification:

```
INT3 =
requirement
    sort    s
    var x y z:  s
    op  Inv:  s -> s
    op  _ @ _ :  s s -> s
    ax  (x @ y) @ z = x @(y @ z)
    cn  N: s
    ax  x @ N = x
    ax  N @ x = x
    ax  x @ Inv(x) = N
    ax  Inv(x) @ x = N
end
```

Consider the algebra A″ with the same carrier and the same interpretations of constant and operation symbols of A′ and the following interpretation of the operation symbol `Inv`:

$\texttt{Inv}^{A''}: \mathbb{Z} \to \mathbb{Z}$
$\texttt{Inv}^{A''}(x) = -x;$

and the algebra B″ similar to a B′ with the following interpretation of the operation symbol `Inv`:

$\texttt{Inv}^{B''}: \{0,1\} \to \{0,1\}$
$\texttt{Inv}^{B''}(0) = 0, \qquad \texttt{Inv}^{B''}(1) = 1$

The algebras A″, B″ are, modulo isomorphism, two abstract data types of the class groups, but they are not the same abstract data type since they are not isomorphic.

In this case the function between specifications α_2, used for proving that `INT3` implements `INT2`, just hides the unary operation "`Inv`" added to `INT2`.

Until now we have used the loose approach to the specifications, since we are interested to representing all the abstract data types which are groups respect to an operation binary, i.e. which have a non empty set of elements, an associative binary operation defined on the elements of such set having a element neutral and the inverse. These general requirements do not specify, for example, as the elements of the set aare made.

If we want to determine a unique abstract data type, for example the additive group $\mathbb{Z}$, we need to specify also which are the constructors of the elements and which conditions are satisfied by them and by the operations manipulating them and thus we have to use the initial approach.

We define the specification `INT4`, with initial approach, implementing `INT3`, by adding the constructors of the elements `Pred` and `Succ`, both unary operation symbols and the formulae expressing their properties and these of the other operation symbols. Thus we get the following specification:

```
INT4 =
design
    sort    int
    var x y:  int
    cn  0:  int
    op  Succ, Pred:  int -> int
    ax  Succ(Pred(x)) = x
    ax  Pred(Succ(x)) = x
    op  _ + _ :  int int -> int
    ax  x + 0 = x
    ax  x + Succ(y) = Succ(x + y)
    ax  x + Pred(y) = Pred(x + y)
    op  - _ :  int -> int
    ax  - 0 = 0
    ax  - Succ(x) = Pred(- x)
    ax  - Pred(x) = Succ(- x)
end
```

The abstract data type represented by the specification is the class of isomorphism represented by $(\mathbb{Z}, +)$ as required.

In this case the function between specifications α_3, used for showing that INT4 implements `INT3`, changes the sort `int` into `s` the constant `0` into `N`, the operations `+` and `-` respectively in @ and `Inv` and hides `Pred` and `Succ`.

Note that in this specification the axioms of `INT3` are not given explicitly, since they logically follow from those of `INT4`. **End example**

2 Formal models of dynamic systems

In the following we use the generic term "dynamic system" for denoting whatever system evolving in the time; processes and concurrent programs, hardware processors, mechanical/electric devices are concrete examples of dynamic systems.

The aim of this section is to introduce a method for specifying dynamic systems: to do that we need a "model", i.e. some elements definable in precise and formal way which represent such dynamic systems. Currently there are several models for dynamic systems, the SMoLCS methodology uses the labelled transition systems.

2.1 Labelled transition systems for modelling dynamic systems

A labelled transition system models the activity of a dynamic system by describing the possible states (interesting situations) of the system and its possible transitions, i.e. its capabilities of passing from a state to another, specifying, with the labels of the transitions also the interaction of the system with the external (w.r.t. the system) environment during each transition.

A *labelled transition system* (shortly *lts*) is a 4-uple

$$(STATE, LABEL, TRANS, s_0)$$

where $STATE$ and $LABEL$ are two sets, whose elements are respectively the states and the labels of the system, $s_0 \in STATE$ is the initial state, and

$$TRANS \subseteq STATE \times LABEL \times STATE$$

is the transition relation. A triple $(s, l, s') \in TRANS$ represents a transition of the system and it is usually written in the following way: $s \xrightarrow{l} s'$, for suggesting the idea of transition.

The activity of a dynamic system may be described by an lts where:

- the states represent the "interesting" situations that may be reached by the system during its life;
- the transitions represent the *capabilities* of the system (note that we speak of capabilities) of passing from a state (situation) to another one;
- the label of a transition represents the "interaction" of the system with the external (w.r.t. the system) environment while it passes from a state to another one; here "interaction" means both the condition on the external environment for the capability to become effective and the transformation of the external environment due to the execution of the transition;
- the initial state represents the initial situation of the system.

Thus a transition $s \xrightarrow{l} s'$ of an lts has the following meaning: the system in the state (situation) s has the capability of passing in the state (situation) s' interacting with the external environment as represented by the label l.

Example 2.1 A process modelled by an lts
We consider the particular dynamic system made by the following process p, described using a simple concurrent programming language CL, where the processes interact by handshaking communications (by executing the **receive** and **send** commands).

process p
local var x; - - each variable has type natural
begin
 receive x **from** p_1;
 if $x > 0$ **then send** 3 **to** p_2 **else** ($x := 4$ **or** $x := 5$) - - **or** nondeterministic choice
end

Now we try to get an lts formally describing the process p.

The "interesting" situations occurring during the life of p are characterized by the list of the commands still to be executed and by the current value of the local variable x (which may be also undefined); thus

$$P\_STATE = P\_COM \times (\mathbb{N} \cup \{undef\}),$$

where $P\_COM = \{c_0; c_1, c_1, c_2\}$ with
$c_0 =$ **receive** x **from** p_1;
$c_1 =$ **if** $x > 0$ **then send** 3 **to** p_2 **else** ($x := 4$ **or** $x := 5$) and $c_2 =$ **skip**.

The initial situation of p, ps_0 is represented by the pair $(c_0; c_1, undef)$.

The possible interactions of p with the external environment, consisting of the process components of the program in parallel with it, are the exchange of natural numbers with them; thus

$$P\_LABEL = \{p\ SENDS\ n\ TO\ p_2, p\ RECS\ n\ FROM\ p_1 \mid n \in \mathbb{N}\} \cup \{NULL\};$$

NULL correspond to no interaction with the external environment (internal activity).

The capabilities of p in the state ps_0 of receiving a value from the process p_1 are given by the transitions:
$ps_0 \xrightarrow{p\ RECS\ n\ FROM\ p_1} (c_1, n)$ for each $n \in \mathbb{N}$.

Note that p in the initial state has infinite capabilities of action and that the capability corresponding to the natural number $\underline{n}$ will become effective only when p is put in an environment in which there is a process p_1 ready to send $\underline{n}$ to p. For these transitions the label represents a condition on the external environment.

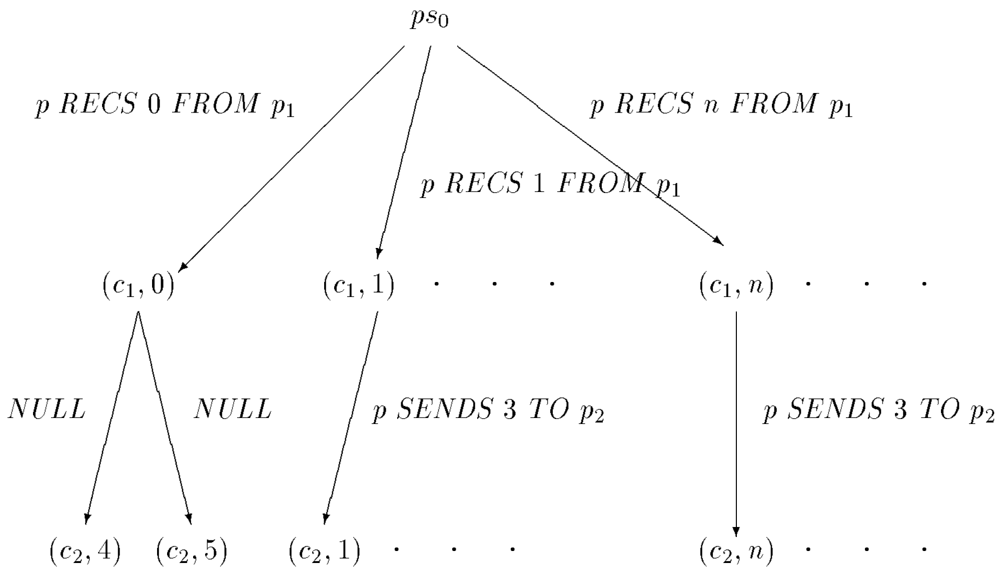
The capability of p in a state (c_1, n) with $n \in \mathbb{N}$, $n > 0$, of sending 3 to p_2 is represented by the transition
$(c_1, n) \xrightarrow{p\ SENDS\ 3\ TO\ p_2} (c_2, n)$.
Also in this case the label "p *SENDS* 3 *TO* p_2" represents a condition on the external environment: "the process p_2 must be ready to receive the value 3 from p".

While p in the state $(c_1, 0)$ has two capabilities of action represented by the transitions: $(c_1, 0) \xrightarrow{NULL} (c_2, 4)$ and $(c_1, 0) \xrightarrow{NULL} (c_2, 5)$.

These transitions do not require any kind of interaction with the external environment and so have a null label; notice also that the state $(c_1, 0)$ represents a nondeterministic situation since the choice between the two possible transitions is nondeterministic.

Moreover in the states (c_2, n), $n \in \mathbb{N}$, there are no transitions.

The set of transitions P_TRANS of the lts modelling p are graphically represented by the following labelled tree (transition tree):



Thus the lts modelling p is

$$P\_LTS = (P\_STATE, P\_LABEL, P\_TRANS, ps_0).$$

As further example, we consider the process p_3 with also shared variables expressed in the same language CL:

process p_3
local var $x := 4$;
shared var y; - - a variable shared with the other processes
begin $y := x$ **end**

Analogously to the previous example, p_3 may be represented by an lts with
$P\_STATE = \{y := x, \textbf{skip}\} \times \{4\}$,
$P\_LABEL = \{WRITES(y, 4)\}$ and
$ps_0 = (y := x, 4)$;
i.e. in this case the external environment includes the shared variable y and the label $WRITES(y, 4)$ represents a transformation of such environment (the value of y is changed into 4).

Moreover this lts has only a transition: $(y := x, 4) \xrightarrow{WRITES(y,4)} (\textbf{skip}, 4)$.
If we replace the command "$y := x$" with "$x := y$", then
$P\_STATE = \{x := y, \textbf{skip}\} \times \mathbb{N}$,

$P\_LABEL = \{READS(y,n) \mid n \in \mathbb{N}\}$ and
$ps_0 = (x := y, 4)$,
where the label "$READS(y,n)$" represents a condition on the external environment (is the value of y equal to n?) and the process p_3 has infinite transitions:
$(x := y, 4) \xrightarrow{READS(y,n)} (\mathbf{skip}, n)$ for each $n \in \mathbb{N}$. **End example**

We have seen how to model a dynamic system with an lts. Instead, when we want to model a class of similar dynamic systems (for example all the processes of CL) that differ only for the initial state, described by the lts's $LTS_i = (STATE, LABEL, TRANS, s_i)$ with $i \in I$, we may forget the initial state and consider a unique lts without initial state, which in the following will still call lts. We observe that in this case each state correspond to a particular dynamic system, whose initial state is such state.

Example 2.2 The processes of the language CL modelled by an lts
We represent all processes, which may be expressed by the simple concurrent programming language CL introduced in Ex. 2.1 by an lts. Each process of the language CL has the following form:

process p - - p is the name of the process
local var $x_1, \ldots, x_n$; - - local variables $(n \geq 0)$
shared var $y_1, \ldots, y_m$; - - variables shared with the other processes $(m \geq 0)$
- - each variable has type natural
begin list of commands **end**

The possible commands are: **skip**, ":=", ";", **if**, **while**, **send**, **receive**, **write**, **read** and **or**.

Now we have to define an lts in which the "interesting" situations occurring during the activity of the processes are characterized by the name of the process, the commands still to be executed and the association of the values to the local variables (also no one).

- The states of the lts are triples in which the first element is the name of the process, the second is the list of the commands still to be executed and the third is the association of the values to the local variables; thus:

$$STATE = PID \times COMMAND \times (VID \to \mathbb{N})_{partial},$$

  with

  - PID set of the process names;
  - VID set of the local variable names;
  - $SVID$ set of the shared variable names;
  - EXP set of the expressions with variables in VID and natural values;
  - $COMMAND$ set of the possible commands, defined as follows:
    * **skip** $\in COMMAND$ (null command);
    * $x := e \in COMMAND$, if $x \in VID$ and $e \in EXP$ (assignment);

- $*$ **if** $e_1 > e_2$ **then** c_1 **else** $c_2 \in \mathit{COMMAND}$, if $e_1, e_2 \in \mathit{EXP}$ and $c_1, c_2 \in \mathit{COMMAND}$ (conditional);
- $*$ **while** $e_1 = e_2$ **do** $c \in \mathit{COMMAND}$, if $e_1, e_2 \in \mathit{EXP}$ and $c \in \mathit{COMMAND}$ (cycle);
- $*$ $c_1; c_2 \in \mathit{COMMAND}$, if $c_1, c_2 \in \mathit{COMMAND}$ (concatenation);
- $*$ c_1 **or** c_2, if $c_1, c_2 \in \mathit{COMMAND}$ (nondeterministic choice);
- $*$ **receive** x **from** p, **send** x **to** $p \in \mathit{COMMAND}$, if $x \in \mathit{VID}$ and $p \in \mathit{PID}$ (receiving/sending a message);
- $*$ **write**(x, y), **read**$(x, y) \in \mathit{COMMAND}$, if $x \in \mathit{VID}$ e $y \in \mathit{SVID}$ (writing/reading a shared variable).

The nondeterministic choice is commutative and associative and the null command concatenated with whatever other command c is the command c (**skip**; $c = c$).

- $(\mathit{VID} \to \mathbb{N})_{partial}$ is the set of all partial functions from the set VID of the local variable names into that of the natural numbers and represents the states of the local memory.
- Eval: $\mathit{EXP} \times (\mathit{VID} \to \mathbb{N})_{partial} \to \mathbb{N}$ expression evaluation function, given an expression and a state of the local memory returns the expression value.

- The possible interactions of the CL processes with the external environment, consisting of the processes in parallel with them and of the shared variables to whom they may access, are the exchange of natural numbers with other processes and the reading and writing of the shared variables; thus

$$\mathit{LABEL} = \{p\ \mathit{SENDS}\ n\ \mathit{TO}\ p', p\ \mathit{RECS}\ n\ \mathit{FROM}\ p' \mid n \in \mathbb{N}, p, p' \in \mathit{PID}\} \cup \{\mathit{WRITES}(y, n), \mathit{READS}(y, n) \mid n \in \mathbb{N}, y \in \mathit{SVID}\} \cup \{\mathit{NULL}\}.$$

- The set of transitions of the lts modelling the processes is given analogously to the previous example, by considering all possible states and all possible interactions with the external environment in such states. TRANS is inductively defined as follows: for each $p, p' \in \mathit{PID}$; $e, e_1, e_2 \in \mathit{EXP}$; $x \in \mathit{VID}$; $y \in \mathit{SVID}$; $c, c', c_1 \in \mathit{COMMAND}$; $l \in \mathit{LABEL}$; $n \in \mathbb{N}$ and $lm \in (\mathit{VID} \to \mathbb{N})_{partial}$
  - assignment

    $(p, x := e, lm) \xrightarrow{\mathit{NULL}} (p, \mathbf{skip}, lm[\mathit{Eval}(e, lm)/x]) \in \mathit{TRANS}$
  - conditional

    $(p, \mathbf{if}\ e_1 > e_2\ \mathbf{then}\ c\ \mathbf{else}\ c', lm) \xrightarrow{\mathit{NULL}} (p, c, lm) \in \mathit{TRANS}$, if $\mathit{Eval}(e_1, lm) > \mathit{Eval}(e_2, lm)$

    $(p, \mathbf{if}\ e_1 > e_2\ \mathbf{then}\ c\ \mathbf{else}\ c', lm) \xrightarrow{\mathit{NULL}} (p, c', lm) \in \mathit{TRANS}$, if $\mathit{Eval}(e_1, lm) \leq \mathit{Eval}(e_2, lm)$
  - cycle

    $(p, \mathbf{while}\ e_1 = e_2\ \mathbf{do}\ c, lm) \xrightarrow{\mathit{NULL}} (p, c; \mathbf{while}\ e_1 = e_2\ \mathbf{do}\ c, lm) \in \mathit{TRANS}$, if $Eval(e_1, lm) = Eval(e_2, lm)$

$(p, \mathbf{while}\ e_1 = e_2\ \mathbf{do}\ c, lm) \xrightarrow{NULL} (p, \mathbf{skip}, lm) \in TRANS$,

if $Eval(e_1, lm) \neq Eval(e_2, lm)$

– concatenation

$(p, c; c_1, lm) \xrightarrow{l} (p, c'; c_1, lm') \in TRANS$, if $(p, c, lm) \xrightarrow{l} (p, c', lm') \in TRANS$

– nondeterministic choice

$(p, c\ \mathbf{or}\ c_1, lm) \xrightarrow{l} (p, c', lm') \in TRANS$, if $(p, c, lm) \xrightarrow{l} (p, c', lm') \in TRANS$

– reception /sending

$(p, \mathbf{receive}\ x\ \mathbf{from}\ p', lm) \xrightarrow{p\ RECS\ n\ FROM\ p'} (p, \mathbf{skip}, lm[n/x]) \in TRANS$

$(p, \mathbf{send}\ x\ \mathbf{to}\ p', lm) \xrightarrow{p\ SENDS\ lm(x)\ TO\ p'} (p, \mathbf{skip}, lm) \in TRANS$

– reading/writing a shared variable

$(p, \mathbf{read}(x, y), lm) \xrightarrow{READS(y,n)} (p, \mathbf{skip}, lm[n/x]) \in TRANS$

$(p, \mathbf{write}(x, y), lm) \xrightarrow{WRITES(y,lm(x))} (p, \mathbf{skip}, lm) \in TRANS$

End example

2.2 Concurrent systems (structured dynamic systems)

A particular class of lts's, said *concurrent systems*, may be used to represent structured dynamic systems (i.e. groups of dynamic systems interacting among them). In this case the states of the lts represent the various situations occurring during the life of the group of systems and the transitions represent their whole capabilities of action.

For giving a concurrent system we have to specify the dynamic systems interacting among them (said the *active components or processes* of the system); these systems are in turn represented by lts's.

In general these active components may also interact indirectly among them, for example, writing and reading a shared memory, putting and taking messages from some global buffer and so long; thus we assume that the concurrent systems have also *passive components*. The difference between active and passive components is that the second may change their states only as result of an action of some active component. The possible states of the passive components are represented by a set of values.

A situation of a concurrent system is thus characterized by the situations of its components (active and passive) and by how they are put together for giving the whole system.

The activity of a concurrent system consists of simultaneous executions of actions by some of the active components (and of the corresponding transformations of the passive components), thus its transitions are defined by rules having form

if $a_1 \xrightarrow{l_1} a'_1$ **and** ... **and** $a_n \xrightarrow{l_n} a'_n$ **and** $cond(syst, l_1, \ldots, l_n)$ **and**
$a_1, \ldots, a_n$ are components of $syst$
then $syst \xrightarrow{l} syst'$ with $a'_1, \ldots, a'_n$ components of $syst'$

where:

- $a_1, \ldots, a_n$ are the states of the active components generating the transition,
- *syst* is a state of the concurrent system, it contains $a_1, \ldots, a_n$ and may contain passive components and other active ones which do not take part in the transition,
- $syst'$ is the state of the system after having executed the transition whose interaction with the external environment is represented by the label l, it should include the active components $a'_1, \ldots, a'_n$, but in $syst'$ some components (active and passive) of *syst* may have disappeared, some new one may be added and some passive one may have changed their states.

The meaning of such rules should be clear: if $a_1 \xrightarrow{l_1} a'_1, \ldots, a_n \xrightarrow{l_n} a'_n$ are transitions of the lts describing the active components and the boolean expression $cond(\ldots)$ is true, then $syst \xrightarrow{l} syst'$ is a transition of the concurrent system.

Usually the states of the concurrent systems are represented as multisets of states of the components; in such case the previous rules have the form:

if $a_1 \xrightarrow{l_1} a'_1$ **and** ... **and** $a_n \xrightarrow{l_n} a'_n$ **and**
$cond(a_1 \mid \ldots \mid a_n \mid ma_1 \mid ma_2 \mid p_1 \mid \ldots \mid p_h \mid mp_1 \mid mp_2, l_1, \ldots, l_n)$
then
$$a_1 \mid \ldots \mid a_n \mid ma_1 \mid ma_2 \mid p_1 \mid \ldots \mid p_h \mid mp_1 \mid mp_2 \xrightarrow{l} a'_1 \mid \ldots \mid a'_n \mid ma_1 \mid ma_3 \mid p'_1 \mid \ldots \mid p'_h \mid mp_1 \mid mp_3$$

where:

- $n \geq 1$ and $h \geq 0$;
- $a_1, \ldots, a_n$ are the active components which perform a transition changing state;
- ma_1, ma_2, ma_3 are the multisets ofactive components that respectively stay idle, are deleted and are created;
- $p_1, \ldots, p_h$ are the passive components which are modified;
- mp_1, mp_2 and mp_3 are the multisets of passive components which respectively are unchanged, deleted and created.

Example 2.3 The CL programs modelled by a concurrent lts
We give the concurrent system *C_LTS* describing the programs of the simple concurrent language CL introduced in Ex. 2.1 and 2.2; such programs have form

shared var $id_1, \ldots, id_n$;
process ...;
...
process ...;

The processes of a program perform their activity in interleaving way except when they perform the handshaking communications.

Let $P\_LTS = (P\_STATE, P\_LABEL, P\_TRANS)$ be the lts describing the activity of the CL processes defined in Ex. 2.2.

C_LTS has several active components, one for each process in the program; and it has a passive component: the shared memory, whose states are represented by partial functions from variable identifiers ($SVID$) into natural numbers. Thus, the set of the states of C_LTS is

$$C\_STATE = SET \times (SVID \rightarrow \mathbb{N})_{partial}$$

with $SET \subseteq \mathcal{P}(P\_STATE)$ and such that the elements of SET correspond to groups of processes with distinct identifiers.

The processes may only interact among them and with the shared memory, thus C_LTS has not interactions with the external (to the concurrent system) environment, we say that it is a closed system, thus $C\_LABEL = \{NULL\}$.

The transitions of C_LTS are given by the following rules, where $p, p' \in PID$; $y \in SVID$; $ps_1, ps'_1, \ldots \in P\_STATE$; $v \in \mathbb{N}$; $sm \in (SVID \rightarrow \mathbb{N})_{partial}$.

- internal actions of a process

  if $ps_1 \xrightarrow{NULL} ps'_1$

  then $ps_1 \mid ps_2 \mid \ldots \mid ps_n \mid sm \xrightarrow{NULL} ps'_1 \mid ps_2 \mid \ldots \mid ps_n \mid sm \qquad n \geq 1$

- handshaking communication

  if $ps_1 \xrightarrow{p\ SENDS\ v\ TO\ p'} ps'_1$ **and** $ps_2 \xrightarrow{p'\ RECS\ v\ FROM\ p} ps'_2$

  then $ps_1 \mid ps_2 \mid ps_3 \mid \ldots \mid ps_n \mid sm \xrightarrow{NULL} ps'_1 \mid ps'_2 \mid ps_3 \mid \ldots \mid ps_n \mid sm \qquad n \geq 2$

- writing a shared variable

  if $ps_1 \xrightarrow{WRITES(y,v)} ps'_1$

  then $ps_1 \mid ps_2 \mid \ldots \mid ps_n \mid sm \xrightarrow{NULL} ps'_1 \mid ps_2 \mid \ldots \mid ps_n \mid sm[v/y] \quad n \geq 1$

  where $sm[v/y]$ denotes the function s.t. $sm[v/y](y) = v$ and for each $y' \neq y$ $sm[v/y](y') = sm(y')$

- reading a shared variable

  if $ps_1 \xrightarrow{READS(y,v)} ps'_1$ **and** $sm(y) = v$

  then $ps_1 \mid ps_2 \mid \ldots \mid ps_n \mid sm \xrightarrow{NULL} ps'_1 \mid ps_2 \mid \ldots \mid ps_n \mid sm \qquad n \geq 1$

End example

In general the number of components of a concurrent system is not fixed but it may dynamically vary, i.e. new components may be created and/or other components may terminate their activity and dissappear; creations and terminations are caused by the execution of some action by some active component.

Example 2.4 Adding the commands **abort** and **create** to CL
Assume to add to the programming language CL the commands for aborting a process and for creating a new process; thus we need to add to the lts representing the processes:

- the commands **abort** p and **create** ps,
- the labels $\{ABORTS(p), CREATES(ps) \mid p \in PID, ps \in P\_STATE\}$
- the transitions
  $(p, \textbf{abort}\ p'; c, lm) \xrightarrow{ABORTS(p')} (p, c, lm), \quad (p, \textbf{create}\ ps; c, lm) \xrightarrow{CREATES(ps)} (p, c, \ldots);$

and to the concurrent system representing the programs the transitions defined by:

$$\begin{array}{l}\textbf{if } ps_1 \xrightarrow{ABORTS(p)} ps_1' \textbf{ and } Name(ps_2) = p\\ \textbf{then } ps_1 \mid ps_2 \mid \ldots \mid ps_n \mid sm \xrightarrow{NULL} ps_1' \mid ps_3 \mid \ldots \mid ps_n \mid sm \quad \text{with } n \geq 2\end{array}$$

$$\begin{array}{l}\textbf{if } ps_1 \xrightarrow{CREATES(ps)} ps_1'\\ \textbf{then } ps_1 \mid ps_2 \mid \ldots \mid ps_n \mid sm \xrightarrow{NULL} ps_1' \mid ps \mid ps_2 \mid \ldots \mid ps_n \mid sm \quad \text{with } n \geq 1.\end{array}$$

End example

2.3 A standard schema as a guide for defining concurrent systems

The activity of a concurrent system consists in the simultaneous executions of actions by some of the active components (and of the corresponding trasformations of the passive components) modelled by the transitions, which are defined by rules having the general form introduced in Sect. 2.2. In general the definition of the transitions of a concurrent systems is very complex and it is advisable to proceed in a modular way starting from description of partial actions which are, step after step, composed with other partial actions until we get the global activity of the system.

Thus, we introduce a standard schema for specifying concurrent systems whose states are characterized by a multiset of active components and by a unique passive component (standard systems) and whose transitions are defined in three standard steps. In the first step, *synchronization*, we define which groups of actions of the components are synchronous, i.e. which groups of actions must happen at the same time since in some way complementary, as for example the sending and the reception of a message (handshaking communication); in the second step, *parallelism*, we define which synchronous actions may happen together, i.e. may be executed in parallel and in the third step, *monitoring*, we define which groups of synchronous actions in parallel (given in the previous step) become final actions of the system respect to some global conditions on it.

The first two steps are formally defined by giving two auxiliary concurrent systems with the same active and passive components of the final system.

The form of the rules for each step is as follows.

Synchronization The rules for the synchronization step have the following form:

if $a_1 \xrightarrow{l_1} a'_1$ **and** ... **and** $a_n \xrightarrow{l_n} a'_n$ **and** $cond(l_1, \dots, l_n, p)$

then $a_1 \mid \dots \mid a_n \mid p \xrightarrow{sl} a'_1 \mid \dots \mid a'_n \mid p'$ with $n \geq 1$

The intuitive meaning of a rule having such form is:
each time $cond(\dots)$ holds, the actions of the active components

$$a_1 \xrightarrow{l_1} a'_1 \ \dots \ a_n \xrightarrow{l_n} a'_n$$

may be synchronized given a transition labelled by sl, where the passive component p is changed into p':

$$a_1 \mid \dots \mid a_n \mid p \stackrel{sl}{\Longrightarrow} a'_1 \mid \dots \mid a'_n \mid p'.$$

Parallelism The rules for the parallelism step have form:

1. **if** $ma \mid p \stackrel{sl}{\Longrightarrow} ma' \mid p'$ **then** $ma \mid p \stackrel{sl}{\sim\!\sim\!\sim\!>} ma' \mid p'$
   The meaning of such rule is that each synchronous action is a parallel action.
2. **if** $ma_1 \mid p \stackrel{pl_1}{\sim\!\sim\!\sim\!>} ma'_1 \mid p'$ **and** $ma_2 \mid p \stackrel{pl_2}{\sim\!\sim\!\sim\!>} ma'_2 \mid p''$ **and** $cond(pl_1, pl_2)$
   then $ma_1 \mid ma_2 \mid p \stackrel{pl_1}{\sim\!\sim\!\sim\!>} ma'_1 \mid ma'_2 \mid p'''$
   The meaning of a rule having such form is:
   each time $cond(pl_1, pl_2)$ holds, the two transitions

$$ma_1 \mid p \stackrel{pl_1}{\sim\!\sim\!\sim\!>} ma'_1 \mid p', \quad ma_2 \mid p \stackrel{pl_2}{\sim\!\sim\!\sim\!>} ma'_2 \mid p''$$

   may be performed in parallel giving the transition

$$ma_1 \mid ma_2 \mid p \stackrel{pl_1}{\sim\!\sim\!\sim\!>} ma'_1 \mid ma'_2 \mid p'''$$

   In general the transformation of the passive component from p into p''' depends on the transformations made by the two transitions labelled by pl_1 and pl_2.

Monitoring The rules for the monitoring step have form:

if $ma \mid p \stackrel{pl}{\sim\!\sim\!\sim\!>} ma' \mid p'$ **and** $cond(pl, \underline{ma}, p)$

then $ma \mid \underline{ma} \mid p \xrightarrow{l} ma' \mid \underline{ma} \mid p'$

The meaning of a rule having such form is:

each time the condition $cond(\dots)$ holds, the parallel action

$$ma \mid p \stackrel{pl}{\sim\!\sim\!\sim\!>} ma' \mid p'$$

is allowed by the monitor and becomes an action of the system represented by the transition

$$ma \mid \underline{ma} \mid p \xrightarrow{l} ma' \mid \underline{ma} \mid p'$$

($\underline{ma}$ is the multiset of active components which do not take part in the action).

3 Specifications of dynamic systems

The basic idea is that of combining the concepts introduced in Sect. 1 on the specifications of the data types with that introduced in Sect. 2 to specify formally and to make more abstract the lts's by representing states, labels and values as abstract data types.

We start by introducing dynamic data types, i.e. data types with sorts of dynamic elements modelled by lts's: the elements of a dynamic sort *ds* represent the states of an lts; there exists an associated sort of labels *l-ds* for representing the labels of the lts and a predicate "$\_ \xrightarrow{\_} \_ : ds\ l\text{-}ds\ ds$" for representing the transition relation of the lts.

Since the concrete/abstract dynamic data types are the natural extension of the static concrete/abstract data types to include also dynamic elements, all concepts and considerations of Sect. 1.2 are still valid; we emphasize only the difference of the two approaches to the specification of a dynamic system with respect to the activity of the same.

Analogously to Sect. 1.2, also for the specification of a dynamic system there are two kinds of approaches:

- initial approach and
- loose approach.

The initial approach determines a particular typology of dynamic system (an abstract dynamic data type), whose activity is completely described by the axioms about the predicate $\longrightarrow$ representing the transition relation of the lts.

The loose approach, instead, determines various typologies of dynamic system (classes of abstract dynamic data types), which satisfy the properties on the system activity expressed by the axioms about the predicate $\longrightarrow$, representing the transition relation of the lts. We use this kind of approach when we want to express very general properties on the activity of the system, as already seen in Sect. 1.2.

Notice that for expressing very general dynamic properties, e.g.: eventually a certain situation will be reached, a property holds until another one holds and so long, the first-order logic is not more adequate instead we need to use temporal combinators, which will be introduced in Sect. 3.3.

Finally, for expressing properties on the concurrent strucutre of the system we use a subclass of the dynamic data types, the entity one, which will be introduced in Sect. 3.4.

3.1 Concrete dynamic data types as dynamic algebras

The dynamic algebras are particular algebras (see Sect. 1.1) in which some sorts correspond to dynamic elements and thus that may be considered *concrete dynamic data types*. We use special signatures which have, other than the usual static sorts, explicit *dynamic sorts*, of dynamic elements modelled by an lts. Moreover for each dynamic sorts *ds* the signature has:

- a sort *l-ds*, for the *labels* of the transitions of the associate lts;
- a predicate $\_ \xrightarrow{\_} \_ : ds\ l\text{-}ds\ ds$, for representing the transition relation of the associate lts.

Finally a dynamic algebra on a dynamic signature is simply an algebra on such signature.

If DA is a dynamic algebra with dynamic signature $D\Sigma$ and ds is a dynamic sort of $D\Sigma$, then the elements of sort ds, the elements of sort $l\text{-}ds$ and the interpretation of the predicate $\longrightarrow$ are respectively the states, the labels and the transition relation of an lts which describes the dynamic elements of sort ds.

The formal definition of dynamic signature and algebra is given in Fig. 7.

- A *dynamic signature* $D\Sigma$ is a pair (Σ, DS) where:
  - $\Sigma = (S, CN, OP, PR)$ is a signature,
  - $DS \subseteq S$ (the elements in DS are the *dynamic sorts*),
  - for each $ds \in DS$ there exist a sort $l\text{-}ds \in S$ and a predicate $\_ \xrightarrow{\_} \_ : ds\ l\text{-}ds\ ds \in PR$.
- A *dynamic algebra* DA on $D\Sigma$ (or *$D\Sigma$-algebra*) is an algebra on the signature Σ.

Figure 7: Dynamic signature and algebra.

The specification language for describing dynamic signatures is analogous to that introduced in Sect. 1.1.1 for the static signatures, enriched with the key word **dsort** which must precede the symbols of the dynamic sorts (see [?]).

Example 3.1 Buffers containing natural values organized as stacks (last in first out) We give a formal model of the buffers by means of an lts formalized with a dynamic algebra. Consider the following dynamic signature

```
sortnat
cn  0:  nat
op  Succ:  nat -> nat
dsort buf:  _ -- _ --> _
cn  Empty:  buf
op  Put:  nat buf -> buf
op  First:  buf -> nat
op  Get:  buf -> buf
op  I,O: nat -> lab_buf
pr  Is_Empty:  buf
```

`nat` is a static sort, while `buf` is a dynamic sort; thus implicitly we have also the sort `lab_buf` and the predicate `_ -- _ --> _`. The interactions with the external, w.r.t. the buffer, environment, i.e. the reception and the returning of a value, are represented by the two operations `I` and `O`.

The buffers are modelled by the ΣBUF-dynamic algebra BUF, where:

- $\mathtt{nat}^{\mathrm{BUF}} = \mathbb{N} \cup \{\#nat\}$.
- $\mathtt{buf}^{\mathrm{BUF}}$ and the interpretation of the operations `Empty`, `Put`, `First` and `Get` are respectively the set of the stacks of natural numbers (plus error) and the usual operations empty stack, put an element at the top to the stack, the first element of the stack and get the first element from the stack.
- The interpretation of the predicate `Is_Empty` is the subset of the stacks consisting of the unique element empty stack; `OKbuf` and `OKnat` (always present in each signature) are respectively the set of the stacks and of the natural numbers.
- If we assume that the buffers are bounded and that may contain at most k elements, then the interpretation of `_ -- _ --> _` in BUF is the relation consisting of the following triples (here and in following the interpretation of a symbol of constant/ operation/predicate `Symb` in BUF, $\mathtt{Symb}^{\mathrm{BUF}}$, is simply denoted by `Symb`):

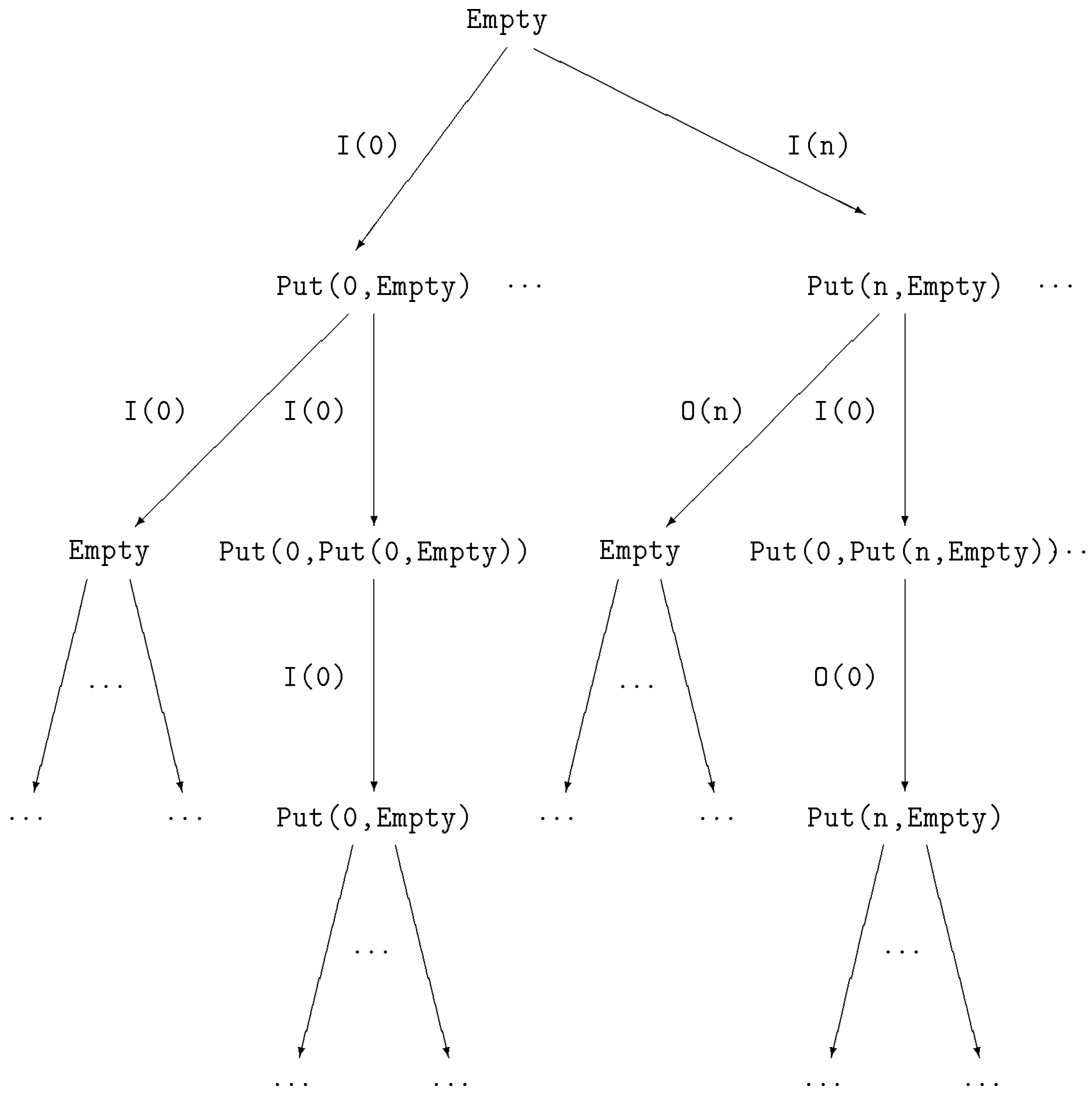
  `b -- I(n) --> Put(n,b)` for each `n` and each `b` with at most k elements,

  `b -- O(First(b)) --> Get(b)` for each `b` s.t. `First(b)` is not error.
- If we assume that the buffers are not bounded, then `_ -- _ --> _` consists of the triples:

  `b -- I(n) --> Put(n,b)` for each `n` and each `b`,

  `b -- O(First(b)) --> Get(b)` for each `b` s.t. `First(b)` is not error.

The activity of a bounded buffer, with $k = 2$, which is initially empty (represented by the term `Empty`) is given by the following tree.



End example

3.2 Specifications of abstract dynamic data types

Since we represent a particular dynamic system by a concrete dynamic data type, we can represent a typology of dynamic systems, admitting different realizations, as an *abstract dynamic data type*, i.e. as a class of isomorphic dynamic algebras which may all be considered concrete realizations (models) of such abstract structure.

An abstract dynamic data type is represented, analogously to a static abstract data type (see Sect. 1.2), with particular algebraic specifications called *dynamic specifications*.

A dynamic specification is a pair consisting of a dynamic signature $D\Sigma$ and a set AX of axioms, i.e. logic formulae, which represent (the static, dynamic and about the concurrent structure) properties of the dynamic data type. A dynamic specification determines a set of concrete structures (dynamic algebras), all that which satisfy the axioms, i.e. which are models of AX.

A dynamic algebra DA is said *model* of a formula ϕ if ϕ is *valid* in DA; moreover a dynamic algebra DA is said *model* of a set of formulae if DA is model of each formula belonging to the set, analogously to the usual static algebras. In particular the definition of first-order logic formulae relative to a dynamic signature $D\Sigma$ and of their validity in a dynamic algebra DA are analogous to the corresponding static definitions given respectively in Fig. 4 and 5 in Sect. 1.2.

Also for the dynamic algebras there are two kinds of approaches to the specification: the initial and the loose, fully analogous with that seen in Sect. 1.2.

Recall that the initial approach chooses among the models of a specification a particular model, said the initial model, which is the *abstract data type defined by the specification*, characterized by the following properties:

- each element in the algebra is represented by a closed term;
- only the identifications between elements induced by the axioms hold;
- an atom is true if and only if its truth logically follows from the axioms.

Notice still that the above properties make that in the initial model only the formulae which logically follow from the axioms hold; thus in the development of an initial specification for a dynamic system we have to completely formalize the activity of the system. In particular thus, the activity of the dynamic system described by the the set of axioms, using the predicate $\longrightarrow$ (representing the transition relation of the lts modelling the system) is the minimum one.

Example 3.2 Specification of the CL processes
We give the specification PROC, with initial semantic, of the dynamic system representing the processes of the concurrent programming language CL used in the examples of Sect. 2.

Notice that, since it is a quite complex example, we use the construct for structuring the specifications `use` (see [?]), which allows to modularly define a dynamic specification, using also parametric specifications as `MAP` (finite maps).

First we give the specifications of the states and of the labels.

The "interesting" situations occurring during the activity of the CL processes are characterized by the commands still to be executed and by an association of values with local variables (which may be also undefined); thus it is possible to represent them by making dynamic the sort corresponding to such triples, i.e. with the specification

```
STATE_PROC =
design
    use PID, COMMAND, MAP(VID,NAT)
    dsort proc:  _ -- _ --> _
    op  < _ _ _ >:  pid com map(vid,nat) -> proc
end
```

where `VID` and `PID`, the specifications of the local variable and of the process identifiers respectively, are not further detailed (we only know that have respectively the sorts `vid` and `pid`) and

```
COMMAND =
design
    use VID, SVID, PID, EXP
    sort    com
    var c c' c'': com
    cn  Skip: com
    op  _ := _ : vid exp -> com
    op  If _ > _ Then _ Else _ :  exp exp com com -> com
    op  While _ = _ Do _ :  exp exp com -> com
    op  _ ; _ :  com com -> com
    ax  Skip ; c = c
    op  _ Or _ :  com com -> com
    ax  c Or c' = c' Or c
    ax  (c Or c') Or c'' = c Or (c' Or c'')
    op  Receive _ From _ ,Send _ To _ :  vid pid -> com
    op  Read, Write:  vid svid -> com
end
```

where `SVID` and `EXP`, the specification of shared variable identifiers and of the expressions with variables in `VID` and natural values respectively, are not further detailed (we only know that `EXP` uses the specification `NAT` and the specification of the partial maps from `VID` into the natural `MAP(VID,NAT)`, and that it has the sort `exp` and an operation corresponding to the function of evaluation of the expressions `Eval: exp map(vid,nat) -> nat`).

The possible interactions of the processes with the external environment, consisting in the processes in parallel with them and the shared variables to whom they may access, are modelled by the following specification using operations which return elements of sort `lab_proc` i.e. belonging to the sort implicitly associated with the dynamic sort `proc`.

```
LABEL_PROC =
design
    use NAT, PID
    sort    lab_proc
    cn  NULL: lab_proc    **   internal action
    **  a process sends to (receive from) a process a natural
    op  _ SENDS _ TO _ ,_ RECS _ FROM _ :  pid nat pid -> lab_proc
    **  a process assigns to (reads from) a shared variable a natural
    op  WRITES,READS: svid nat -> lab_proc
end
```

The transitions of the lts modelling the processes are defined by a set of axioms given in the following[3] about the predicate `_ -- _ --> _: proc lab_procproc` implicitly associated with the dynamic sort `proc`.

```
PROC =
```

[3]For the comments see Ex. 2.2.

```
design
    use STATE_PROC, LABEL_PROC
    var e e1 e2:  exp
    var n:  nat
    var c c' c'' c1:  com
    ax  < p x := e lm > -- NULL --> < p Skip lm[Eval(e,lm)/x)]
    ax  if Eval(e1,lm) > Eval(e2,lm) then
            < p If e1 > e2 Then c Else c' lm > -- NULL --> < p c lm >
    ax  if Eval(e1,lm) <= Eval(e2,lm) then
            < p If e1 > e2 Then c Else c' lm > -- NULL --> < p c' lm >
    ax  if Eval(e1,lm) = Eval(e2,lm) then
            < p While e1 = e2 Do c lm > -- NULL --> < p c ; While e1 = e2 Do c lm >
    ax  if Eval(e1,lm) =/= Eval(e2,lm) then
            < p While e1 = e2 Do c lm > -- NULL --> < p Skip lm >
    ax  if < p c lm > -- l --> < p c' lm' > then
            < p c ; c1 lm > -- l --> < p c' ; c1 lm' >
    ax  if < p c lm > -- l --> < p c' lm' > then
            < p c Or c1 lm > -- l --> < p c' lm' >
    ax  < p Receive x From p' lm > -- p RECS n FROM p' --> < p Skip lm[n/x]
    ax  < p Send x To p' lm > -- p SENDS lm(x) TO p' --> < p Skip lm >
    ax  < p Read(x,y) lm > -- READS(y,n) --> < p Skip lm[n/x]
    ax  < p Write(x,y) lm > -- WRITES(y,lm(x)) --> < p Skip lm >
end
```

End example

Recall that the loose approach determines a class of dynamic abstract data types. In this case a specification expresses the general properties of a dynamic data type, in particular it expresses the properties on the activity of the dynamic system by means of the predicate `_ -- _ --> _`, representing the transition relation of the lts modelling the system, but it does not completely describe the activity of the system as in the initial approach.

Finally notice that for expressing general properties of the activity of a dynamic system, frequently, the first-order logic is not adequate. Indeed, in general, we want to express properties as: the system perform an action until a certain situation will be reached, starting from a given situation eventually a certain action will be executed, the system performs an action and immediately after it performs another one and so long. This kind of properties may be expressed conveniently using the temporal combinators, which will be introduced in Sect. 3.3.

Example 3.3 Specification of the buffers (continuation)
In this example we define by appropriate dynamic specifications with loose semantics various classes of buffers; we try to show which properties could easily expressed with the first-order logic and which could not.

The specification of a class of buffers is a pair made by the dynamic signature ΣBUF given in Ex. 3.1 and by the axioms given in the following.

- properties of the data contained in the buffers

  `OKnat(Succ(n))`

- static properties (organization of the buffers as stacks, the operations `First` and `Get` are not defined on the empty buffer)

  `not OKbuf(Get(Empty))` `Get(Put(n,b)) = b` `First(Put(n,b)) = n`

  `not OKbuf(First(Empty))` `Is_Empty(Empty)` `not Empty(Put(n,b))`

- dynamic properties:
  - safety properties:

    if the buffer `b` returns `n`, then `n` is the first element of `b` and such element is deleted from `b`

    `if b -- O(n) --> b' then n = First(b) and b' = Get(b)`

    if the buffer `b` receives an element, then this element is inserted in `b`

    `if b -- I(n) --> b' then b' = Put(n,b)`

  - liveness properties:

    the buffers have the capability of returning whatever natural number `n` which have received and keep this capability until effectively return it; thus a buffer `b` which receives `n` performs input/output transitions in such a way that in whatever state either `n` has been returned or it is possible to reach another state in which `n` may be returned.

    > if `b0 -- I(n) --> b1` [i.e. `b1 = Put(n,b0)`] then
    > for each $\mathtt{b}_k$ such that
    > `b1 -- l1 --> b2`, `b2 -- l2 --> b3` ... and $\mathtt{b}_{k-1}$ `--` $\mathtt{l}_{k-1}$ `-->` $\mathtt{b}_k$,
    > where the labels `l1`,..., $\mathtt{l}_{k-1}$ are either `O(...)` or `I(...)`
    > either one of the above labels is `O(n)`,
    > or there exist $\mathtt{b'}_1$, ..., $\mathtt{b'}_m$ s.t.
    > $\mathtt{b}_k$ `-- l'0 -->` $\mathtt{b'}_1$ and ... and $\mathtt{b}_{m-1}$ `-- O(n) -->` $\mathtt{b'}_m$.

    This last property is expressed above informally, since for formalizing it we need complex formulae of the infinitary first-order logic (i.e. with either conjunctions or disjunctions of infinite sets of formulae); in the following section we will see as such property may be conveniently expressed using temporal combinators.

End example

3.3 Temporal combinators for specifying the activity of dynamic systems

We have seen from the last property of Ex. 3.3, that the first-order logic is not always adequate for expressing properties on the dynamic activity; this has lead us to introduce temporal combinators for the specifications of the dynamic systems. Now we see how to extend the first-order logic on a dynamic signature with appropriate temporal combinators.

Given a $D\Sigma$-dynamic algebra DA (see Sect. 3.1) and ds one of its dynamic sort, we can represent the whole activity of the (dynamic) elements of sort ds with the maximal labelled paths, i.e. with their maximal sequences of states and labels of the form
$d_0 \xrightarrow{l_0}{}^{\mathrm{DA}}\ d_1 \xrightarrow{l_1}{}^{\mathrm{DA}} d_2 \ldots$.

We denote by $PATH(\mathrm{DA}, ds)$ the set of such paths for the dynamic elements of sort ds, i.e. $PATH(\mathrm{DA}, ds)$ is the set of all the sequences of type (1) o (2):

(1) $d_0\ l_0\ d_1\ l_1\ d_2\ l_2 \ldots d_n\ l_n \ldots$ (infinite path)

(2) $d_0\ l_0\ d_1\ l_1\ d_2\ l_2 \ldots d_n\ n \geq 0$ (finite path),

where for each $n \in \mathbb{N}$: $d_n \in \mathrm{DA}_{ds}$, $l_n \in \mathrm{DA}_{l\text{-}ds}$, $(d_n, l_n, d_{n+1}) \in\ \longrightarrow^{\mathrm{DA}}$ and in (2) for no d, and l: $(d_n, l, d) \in\ \longrightarrow^{\mathrm{DA}}$ (i.e., if a path is finite then there are not transitions starting from its final state).

If $\sigma \in PATH(\mathrm{DA}, ds)$ and $n \in \mathbb{N}$, then

- $S(\sigma)$ denotes the first element of σ;
- $L(\sigma)$ denotes the second element of σ (if there exists);
- $\sigma|_n$ denotes the path $d_n\ l_n\ d_{n+1}\ l_{n+1}\ d_{n+2}\ l_{n+2} \ldots$ (if there exists).

Such considerations suggest to express the dynamic properties of an element d_0 of sort ds, i.e. the properties of the activity of d_0, as properties of the paths starting from d_0, i.e. in an equivalent way, as properties of the states and of the labels of the path. Thus, we want a logic which allows to express the following properties.

The interesting various properties about the dynamic activity of an element d_0 may be formulated following the schema below.

there exists a path from d_0 for each path from d_0	a certain property holds on the path

Thus the our logic should have the combinators:

– $\triangle(td_0, \ldots)$

  ($\triangle(td_0, \ldots)$ holds if for each path starting from the dynamic element represented by td_0 ... holds);

– $\bigtriangledown(td_0, \ldots)$

  ($\bigtriangledown(td_0, \ldots)$ holds if there exists a path starting from the dynamic element represented by td_0 s.t. ... holds);

where td_0 is a term of dynamic sort representing d_0 and "..." stands for a formula expressing a path property.

The path properties follow the schema:

$$\left[\begin{array}{l}\text{in the first point of the path}\\ \text{in the second point of the path}\\ \text{in each point of the path}\\ \text{[except the first]}\\ \text{there exists a point of the path}\\ \text{[different from the first]}\\ \text{until a property become true}\\ \text{[surely it will become true]}\end{array}\right] \quad \left[\begin{array}{l}\text{a certain propety holds on that point}\\ \text{a certain property holds on the path}\\ \text{from that point}\end{array}\right]$$

where *point* means a pair state, next label (for example the pairs (d_0, l_0), (d_1, l_1), ... are the points of the path $d_0 \xrightarrow{l_0}{}^{\mathrm{DA}} d_1 \xrightarrow{l_1}{}^{\mathrm{DA}} d_2 \ldots$); and point property means a property of either the state or of the label of the point (or of both).

Thus our logic should have the combinators:

- $\bigcirc$ ...: in the second point of the path;
- $\blacksquare$ ...: in each point of the path (except the first $\Box$ ...);
- $\blacklozenge$ ...: there exists a point of the path (except the first $\Diamond$ ...);
- ... $\mathcal{WU}$ ...: the first property holds on the path until the second holds (moreover surely the second one will hold ... $\mathcal{U}$ ...);

where ... stands for path properties.

The properties of the first point of a path are simply expressed by formulae expressing a point property.

While for the point properties we have:

- $[\lambda\, x \,.\, \phi(x)]$, where x is a variable of dynamic sort, for expressing a condition on the state of a point and
- $<\lambda\, y \,.\, \phi(y)>$, where y is a variable of sort label, for expressing a condition on the label of a point,

where ϕ is a logic formula.

Clearly we also need the usual combinators of first-order logic ($\forall$, $\exists$, $\wedge$, $\vee$, $\neg$) also for combining formulae expressing conditions on the paths.

The formulae introduced for expressing the properties of the activity of a dynamic element as properties of its paths are formally defined in Fig. 8 while their validity is given in Fig. 9. Notice that the logic defined in Fig. 8 includes the first-order logic used previously.

In the definitions of Fig. 8 and 9 we use only one temporal combinator, i.e. only the $\mathcal{U}$ combinator; since all other temporal combinators may be defined in term of it as it is shown in Fig. 10.

Now we see some examples of properties of a dynamic element represented by the term *dt* of dynamic sort *ds*, expressed using the combinators introduced above.

- dt eventually will perform the interaction represented by lt (term of sort $l\text{-}ds$) with the external environment, i.e.:

  for each path from dt

  there exists a point in the path s.t. its label is lt;

  formally: $\triangle(dt, \blacklozenge <\lambda\ y\ .\ y = lt>)$.

  If we want that such action is not the first one we use the combinator $\diamond$; formally: $\triangle(dt, \diamond <\lambda\ y\ .\ y = lt>)$.

- It can happen that dt will never perform the interaction with the external environment represented by lt, i.e.:

  there exists a path from dt s.t.

  for each point in the path the label is different from lt;

  formally: $\bigtriangledown(dt, \blacksquare <\lambda\ y\ .\ y \neq lt>)$.

  If we allow that at most the first action may be lt we use the combinator $\square$; formally: $\bigtriangledown(dt, \square <\lambda\ y\ .\ y \neq lt>)$.

- At least in a case dt after a step may be in the situation represented by dt'; i.e.

  there exists a path from dt s.t.

  the state of the second point is dt';

  formally: $\bigtriangledown(dt, \bigcirc\ [\lambda\ x\ .\ x = dt'])$.

- At least in a case dt, immediately performs the interaction represented by lt, and passes the situation represented by dt'; i.e.:

  there exists a path from dt s.t.

  the label of the first point is lt and

  the state of the successive point is dt';

  formally: $\bigtriangledown(dt, <\lambda\ y\ .\ y = lt> \ \wedge\ \bigcirc\ [\lambda\ x\ .\ x = dt'])$.

- dt will never execute the interaction with the external environment represented by lt until it will be in the situation represented by dt' (we require that the situation dt' eventually will be reached); i.e.:

  for each path from dt

  on all the points of the path the label is different from lt

  until the state is not equal to dt';

  formally: $\triangle(dt, <\lambda\ y\ .\ y \neq lt>\ \mathcal{U}\ [\lambda\ x\ .\ x = dt'])$.

  If we do not require that the situation dt' is surely reached we use the combinator $\mathcal{WU}$; formally: $\triangle(dt, <\lambda\ y\ .\ y \neq lt>\ \mathcal{WU}\ [\lambda\ x\ .\ x = dt'])$.

– dt in at least a case will reach infinite times the situation represented by dt'; i.e.:

  there exists a path from dt s.t.

  for infinite points in the path the state is dt';

  formally: $\triangledown(dt, \blacksquare \blacklozenge [\lambda\ x\ .\ x = dt'])$.

– From each situation reachable from dt the situation represented by dt' may reached; i.e.

  for each path from dt

  for each state x, on such path,

  there exists a path from x including the state dt';

  formally: $\triangle(dt, \blacksquare [\lambda\ x\ .\ \triangledown (x, \blacklozenge [\lambda\ x'\ .\ x' = dt'])])$.

Let $D\Sigma = (\Sigma, DS)$ with $\Sigma = (S, CN, OP, PR)$ and X a family of variables indexed on S.
The sets of *dynamic formulae* and of *path formulae* of sort $ds \in DS$ on $D\Sigma$ and X, denoted respectively by $F_{D\Sigma}(X)$ and $P_{D\Sigma}(X)$, are inductively defined as follows.
formulae

- $A_\Sigma(X) \subseteq F_{D\Sigma}(X)$
- $\neg\ \phi_1,\ \phi_1 \wedge \phi_2,\ \phi_1 \vee \phi_2,\ \phi_1 \supset \phi_2 \in F_{D\Sigma}(X)$ if $\phi_1, \phi_2 \in F_{D\Sigma}(X)$
- $\forall\ x\ .\ \phi \in F_{D\Sigma}(X)$ if $\phi \in F_{D\Sigma}(X)$, $x \in X$
- $\triangle(t, \pi) \in F_{D\Sigma}(X)$ if t is term of sort ds, $\pi \in P_{D\Sigma}(X)$

path formulae

- $[\lambda\ x\ .\ \phi] \in P_{D\Sigma}(X)$ if $x \in X_{ds}, \phi \in F_{D\Sigma}(X)$
- $[\lambda\ x\ .\ \phi] \in P_{D\Sigma}(X)$ if $x \in X_{l\text{-}ds}$, $\phi \in F_{D\Sigma}(X)$
- $\neg\ \pi_1,\ \pi_1 \wedge \pi_2,\ \pi_1 \vee \pi_2,\ \pi_1 \supset \pi_2 \in P_{D\Sigma}(X)$ if $\pi_1, \pi_2 \in P_{D\Sigma}(X)$
- $\forall\ x\ .\ \pi,\ \exists\ x\ .\ \pi \in P_{D\Sigma}(X)$ if $\pi \in P_{D\Sigma}(X)$, $x \in X$
- $\pi_1\ \mathcal{U}\ \pi_2 \in P_{D\Sigma}(X)$ if $\pi_1, \pi_2 \in P_{D\Sigma}(X)$.

Figure 8: Formulae of temporal logic and path formulae.

Example 3.4 Dynamic properties of the buffers
Consider again the example of the buffers (see Ex. 3.3), and the specification made by

the dynamic signature ΣBUF and by the set of axioms AX previously defined; now we can formalize the last property that before has been expressed only informally as follows.

The buffers have the capability of returning whatever natural number `n` that they have received and keep this capability until they do not effectively return it.

```
if b -- I(n) --> b' then
b' in each case
now and always ([ x .  x = O(n) ] or [ y .  exists y:  now or eventually
[ x .  x = O(n) ]]
```
End example

Let DA be a dynamic algebra on $D\Sigma$ and $V: X \to$ DA an evaluation in DA of the variables in X. We inductively define when a formula $\phi \in F_{D\Sigma}(X)$ *holds in* DA *w.r.t.* V (written DA, $V \models \phi$) and when a formula $\pi \in P_{D\Sigma}(X)$ *holds in* DA *on a path* $\sigma \in PATH(\mathrm{DA}, ds)$ *w.r.t.* V (written DA, $V, \sigma \models \phi$).
We recall that the interpretation of a term t (atom α) in DA respect to V is denoted by $t^{\mathrm{DA},V}$ ($\alpha^{\mathrm{DA},V}$) and that, for a path σ, $S(\sigma)$,$L(\sigma)$,$\sigma_{|i}$ have been defined previously.
formulae

- DA, $V \models \alpha$ iff $\alpha^{\mathrm{DA},V}$ holds
- $\neg\, \phi$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \supset \phi_2$, $\forall\, x \,.\, \phi$ and $\exists\, x \,.\, \phi$ analogously to what we have seen in Fig. 3 of Sect. 1.2
- DA, $V \models \triangle(t, \pi)$ iff t is not error and for each $\sigma \in PATH(\mathrm{DA}, ds)$, with ds sort of t, such that $S(\sigma) = t^{\mathrm{DA},V}$ DA, $V, \sigma \models \pi$

path formulae

- DA, $V, \sigma \models [\lambda\, x \,.\, \phi]$ iff DA, $V[S(\sigma)/x] \models \phi$
- DA, $V, \sigma \models [\lambda\, x \,.\, \phi]$ iff either DA, $V[L(\sigma)/x] \models \phi$ or $L(\sigma)$ is an error
- $\neg\, \pi$, $\pi_1 \wedge \pi_2$, $\pi_1 \vee \pi_2$, $\pi_1 \supset \pi_2$, $\forall\, x \,.\, \pi$ and $\exists\, x \,.\, \pi$ analogously to what we have seen in Fig. 3 of Sect. 1.2
- DA, $V, \sigma \models \pi_1\, \mathcal{U}\, \pi_2$ iff
  there exists $j > 0$ s.t. $\sigma|_j$ is defined, DA, $V, \sigma|_j \models \pi_2$ and
  for each i s.t. $0 < i < j$ DA, $V, \sigma|_i \models \pi_1$

$\phi \in F_{D\Sigma}(X)$ is *valid* in DA (written DA $\models \phi$) iff DA, $V \models \phi$ for each evaluation V.

Figure 9: Interpretation of the formulae of the temporal logic and of the path formulae.

We define the temporal combinators introduced in Sect. 3.3 in terms of the combinator $\mathcal{U}$ of Fig. 7 and give explicitly their interpretations w.r.t. a dynamic algebra DA on $D\Sigma$ and an evaluation in DA of the variables in X, $V\colon X \to$ DA.

– $\Diamond\, \pi =_{\mathrm{def}}$ **true** $\mathcal{U}\ \pi$ (there exists a point in the path, except the first, in which π holds)

  DA, $V, \sigma \models \Diamond\, \pi$ iff there exists $i > 0$ s.t. $\sigma|_i$ is defined and DA, $V, \sigma|_i \models \pi$

– $\Box\, \pi =_{\mathrm{def}} \neg \Diamond \neg\, \pi$ (for each point in the path, except the first, π holds)

  DA, $V, \sigma \models \Box\, \pi$ iff DA, $V, \sigma|_i \models \pi$ for each $i > 0$ s.t. $\sigma|_i$ is defined

– $\blacksquare\, \pi =_{\mathrm{def}} \pi \wedge \Box\, \pi$ and $\blacklozenge\, \pi =_{\mathrm{def}} \pi \vee \Diamond\, \pi$ ($\blacksquare$ and $\blacklozenge$ correspond, respectively, to the combinators $\Box$ and $\Diamond$ including the initial point, or present)

– $\pi_1\ \mathcal{WU}\ \pi_2 =_{\mathrm{def}} \pi_1\ \mathcal{U}\ \pi_2 \vee \Box\, \pi_1$

  ($\mathcal{WU}$ is the combinator weak until; $\pi_1\ \mathcal{WU}\ \pi_2$ holds on a path σ each time that π_1 holds until π_2 holds; the difference with $\mathcal{U}$ is that we do not require that π_2 eventually holds)

– $\bigcirc\, \pi =_{\mathrm{def}}$ **false** $\mathcal{WU}\ \pi$ (either the path is made by only one point, or π holds on the second point of the path)

  DA, $V, \sigma \models \bigcirc\, \pi$ iff either $\sigma|_1$ is not defined or DA, $V, \sigma|_1 \models \pi$

Figure 10: Temporal combinators derived from $\mathcal{U}$.

3.4 Entity specifications of dynamic systems

Consider the problem of modelling dynamic concurrent systems (i.e. where different components interact among them), wanting to pointing out to structural properties, as in the case in which the various components of the system are distinct by an "identity" and where these components may share other components.

For example, if we want to specify a set of concurrent and communicating processes, some of them may be in the same state and thus they are not distinguishable using it, or object-oriented systems which share some components; in these cases the specifications of dynamic systems seen in Sect. 3.3 are not adequate.

The idea is to consider a particular subclass of the dynamic specifications (and thus of the dynamic signatures and dynamic algebras) such that: each dynamic element has an "identity", i.e. the states of the dynamic elements are pairs $state = id\colon body$, where

– the identity is preserved by the transitions and

– the identity is unique (two different components have different "names").

3.4.1 Entity signatures

The entity signatures are particular dynamic signatures which allow to represent dynamic elements with identity, i.e. entity. In an entity signature, given a sort s corresponding to the bodies we have:

– e-s, the sort of the entities with body of type s (briefly entities of type s);

– l-s, the sort of the identities of the entities of type s;

– $\_ : \_ : l\text{-}s\ s \to e\text{-}s$, the operation which given an identity and a body builds the corresponding entity.

Moreover since the entity signatures are particular dynamic signatures we have:

– the sort l-s of the labels and

– the predicate $\_ \xrightarrow{-} \_ : e\text{-}s\ l\text{-}s\ e\text{-}s$ describing the activity of the elements of e-s.

Thus we have the definition reported in Fig. 11.

An *entity signature* is a pair $E\Sigma = (\Sigma, ES)$ where:

- $\Sigma = (S, CN, OP, PR)$ is a signature,
- $ES \subseteq S$, (the elements of ES are the sorts of the entity bodies),
- for each $s \in ES$ there exist the sort e-s, l-s, $l\text{-}s \in S$, an operation $\_ : \_ : l\text{-}s\ s \to e\text{-}s \in OP$ and a predicate $\_ \xrightarrow{-} \_ : e\text{-}s\ l\text{-}s\ e\text{-}s \in PR$.

Figure 11: Entity signature.

The specification language for describing the entity signatures is analogous to that for the dynamic signatures, where the key words `dsort` is replaced by `esort`. We know, indeed, that each entity signature corresponds to a particular dynamic signature where the dynamic sorts are the sorts e-s for each $s \in ES$.

3.4.2 Entity algebras

Given an entity signature $E\Sigma = (\Sigma, ES)$, an $E\Sigma$-entity algebra is a Σ-algebra in which the interpretations of the special sorts, operations and predicates must respect some conditions. For saying formally which Σ-algebras are entity algebras, we need some technical definitions and for clarity we first illustrate them on an example.

Consider a very simple concurrent language in which sequential processes evolve in an interleaving way and interact among them by handshaking communications; `Nil`, `_ . _` and `_ + _` are the combinators for expressing the sequential processes and `_ || _` is the parallel combinator.

We define in the following the entity signature `SC`.

```
esort proc:  _ -- _ --> _       **   processes
esort prog [ _ -- _ --> _  ]      **   programs
cn  Tau, Alpha, Beta, ...:  lab_proc      **   process actions
cn  a, b, ...:  ident_proc
cn  Nil:  proc
op  _ .  _ :  lab_procproc -> proc
op  _ + _ :  proc proc -> proc
cn  A, B, ...:  ident_prog
op  _ :  ent_proc -> prog
op  _ || _ :  prog prog -> prog
op  _ :  lab_proc -> label(prog)
```

Consider a `SC`-algebra CD such that:

– its carriers are subsets of a quotient of the ground terms on `SC` module the congruence generated by the equations corresponding to the fact that `+` and `||` are commutative, associative and that `Nil` and `id: Nil` for each `id` are their identities;

– its constants and operations are defined in the obvious way;

– the predicates corresponding to the transition relations for programs) are defined by the following inductive rules, where interpretation $\mathtt{Symb}^{CD}$ in the algebra CD of a symbol of either constant or operation or predicate `Symb` is simply denoted by `Symb`; analogously the interpretation $\mathtt{t}^{CD}$ of a ground term is simply denoted by `t`;

```
p: proc, ep: ent_proc, id: ident_proc, l:lab_proc, pg: prog, pid: ident_prog.

id: l . p -- l --> id: p

if id: p1 -- l --> id: p1' then id: p1+ p2 -- l --> id: p1'

if ep -- l --> ep' then pid: ep -- l --> pid: ep'

if pid:pg1 -- l --> pid:pg1' then

    pid:pg1 || pg2 -- l --> pid:pg1' || pg2

if pid:pg1 -- l --> pid:pg1' and pid:pg2 -- l --> pid:pg2' then

    pid:pg1 || pg2 -- Tau --> pid:pg1' || pg2'     l =/= Tau
```

Now we see how to determine the concurrent structure of an entity of CD.

`epg = A:(a: Tau . Nil || b: Beta . Nil)` $\in \mathrm{CD}_{\mathtt{ent\_prog}}$
represents an entity of type `prog` with identity `A` and body
`a: Tau . Nil || b: Beta . Nil`,
which has two *subentities* of type `proc`, represented respectively by
`a: Tau . Nil` and `b: Beta . Nil`
organized in parallel. Thus `epg` may be seen as
`A:(_ || _ )(a: Tau . Nil,b: Beta . Nil)`,
where the function

(∗1) $\_||\_ : \mathrm{CD}_{\mathtt{ent\_proc}} \times \mathrm{CD}_{\mathtt{ent\_proc}} \to \mathrm{CD}_{\mathtt{prog}}$

$$e_1, e_2|| \to e_1||e_2$$

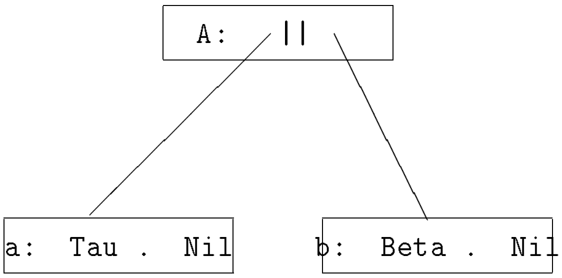
represents the way in which the entities `a: Tau . Nil` and `b: Beta . Nil` are *composed* (i.e., *organized in parallel*) for building the structured entity `epg`.

The entities `a: Tau . Nil` and `b: Beta . Nil` are in some way "simple" (i.e., without components); thus the zero-ary functions

$(*2)$ $\quad Tau.Nil, Beta.Nil \colon \to \mathrm{CD}_{\mathtt{proc}}$

denote that such entities are not given by composing other entities.

Graphically this way of seeing the dynamic structure of `epg` may be represented by the *structure view* of `epg`:



where the term with the holes " || " stands for the function $(*1)$. Note that

`a: Tau . Nil` `b: Beta . Nil`

represent the structure views of the two entities `a: Tau . Nil` and `b: Beta . Nil` (they have no components).

The functions like $(*1)$ and $(*2)$, describing how some entities are put together for getting the body of compound entities, are called *entity composers*; while the structures graphically represented above by graphs are called *structure views of the entities*.

Note that the entity composers are given at a "semantic level" and not at a "syntactic level", i.e. they are compositions of interpretations of operations of the algebra and not just syntactic terms.

Only the constants/operations having result sort s contribute to define the composers for entities of sort e-s, i.e., the structure of the entities is determined by the operations whose result sort is the body sort.

We need also the following terminology; let $E\Sigma = (\Sigma, ES)$, and assume that A is a Σ-algebra, $e \in \cup_{s \in ES} \mathrm{A}_{e\text{-}s}$ and v a structure view, then:

- e has *identity* id iff $e = id{:}\, b$ for some b.
- v is *sound* iff for each subentities e' and e'' of v, if e' and e'' have the same identity, then $e' = e''$.

The formal definition of entity algebra is reported in Fig. 12; note that the last property requires that usually the interpretations in EA of some operations (e.g., the entity composer _: _) are partial functions.

Example 3.5 Examples of entity structures

Consider the entity signature `SC` and the algebra CD on it, previously introduced. We see that:

An $E\Sigma$-*entity algebra* is a Σ-algebra EA such that for each $s \in ES$

- $\mathrm{EA}_{e\text{-}s} \subseteq \mathrm{EA}_{l\text{-}s} \times \mathrm{EA}_s$ (more precisely $\mathrm{EA}_{e\text{-}s}$ is isomorphic to a subset of $\mathrm{EA}_{l\text{-}s} \times \mathrm{EA}_s$) and $(\_ : \_ : l\text{-}s\ s \to e\text{-}s)^{\mathrm{EA}}(id, x) = (id, x)$;
- if $\mathrm{EA} \models e \stackrel{-}{\longrightarrow} -l - - > e'$, then e and e' have the same identity;
- all elements of $\mathrm{EA}_{e\text{-}s}$ have only sound structure views.

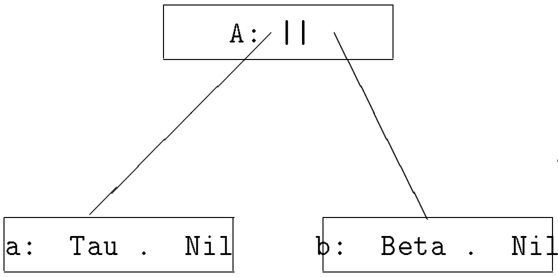
Figure 12: Entity algebra.

Different ways of composing some entities may be equivalent

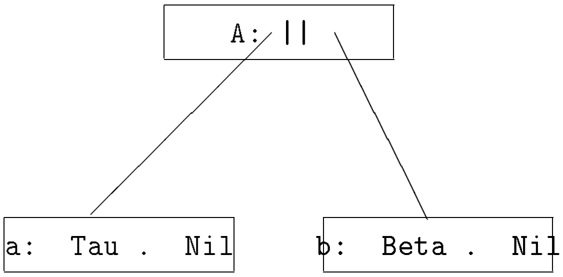
`epg = A:(a: Tau . Nil || b: Beta . Nil)` $\in \mathrm{CD}_{\texttt{ent\_prog}}$
is an entity whose structure may be seen in two different ways; indeed `epg` is also equal to

`A:(b: Beta . Nil || a: Tau . Nil);`

the two structure views of `epg` are graphically represented by



and



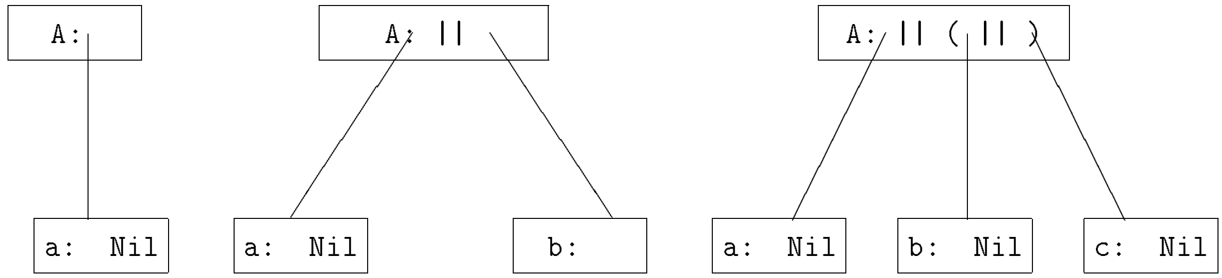
This correspond to the fact that in the CD programs the order of the processes in parallel is not relevant.

Compositions of different groups of entities may be equivalent

It is possible that different structure views of an entity differ also for the number of subentities, as it is shown by the entity `epg' = A: a: Nil`; indeed `epg'` is also equal to

`A:(a: Nil || b: Nil),` `A:(a: Nil || b: Nil || c: Nil)`
and to each number of `Nil` processes put in parallel (recall that the processes having form `id: Nil` are identities for the operation `||`).

Various structure views of `epg'` are graphically represented by:



Thus in the CD programs the processes which cannot perform any action (`id: Nil`) are not semantically relevant.

Not all operations contribute to the entity composers

(i.e. not all operations of the algebra are used for describing the structure of the entities).

Consider a language CD_1 differing from CD only since it has an operation for extracting from a program a process with a certain identity,

`First: ent_progident_proc -> ent_proc`
whose interpretation is given by

`First(pid:(id: p || ep1...|| epn),id) = id: p`

`First(pid:(id1:p1 || ...|| idn: pn),id)` is error if `id =/= idi` for $i = 1, \ldots, n$.

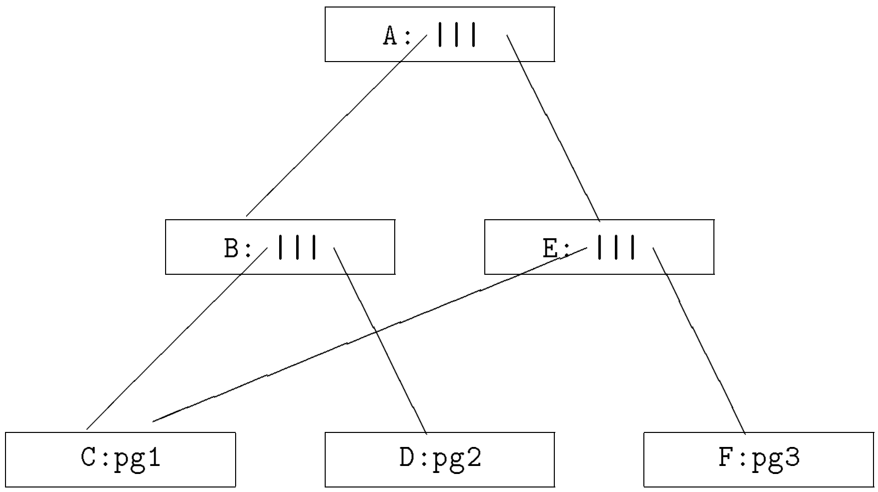
The set of entity composers on CD_1 is the same of that on CD, i.e. the structure of the entities in CD and in CD_1 is the same.

Sharing of subentities

Here we consider a language CD_2 differing from CD only since it has a multilevel parallelism instead of a simple one; we take a new signature `SC2` obtained from `SC` replacing the operation _ `||` _ with

_ `|||`_ : `ent_progent_prog -> prog`,
and give a `SC2`-algebra CD_2 in the same way of CD. In this case an entity of sort `prog` has either a subentity of sort `proc` or two subentities of the same sort `prog`.

`epg'' = A:[ B:(C:pg1 ||| D:pg2) ||| E:(C:pg1 |||F:pg3)]`,
is an entity where the subentity represented by `C:pg1` is shared among the subentities identified by `B` and `E`; its structure is graphically represented by



The entities may terminate and new entities may be created

Here we consider a language CD_3 different from CD only since it allows the termination and the creation of processes. We take a new signature `SC3` obtained from `SC` by adding the operations

`Terminates:   -> lab_proc`   and   `Creates: proc -> lab_proc`,
and CD_3 is defined analogously to CD; the transitions due to the new actions are given by

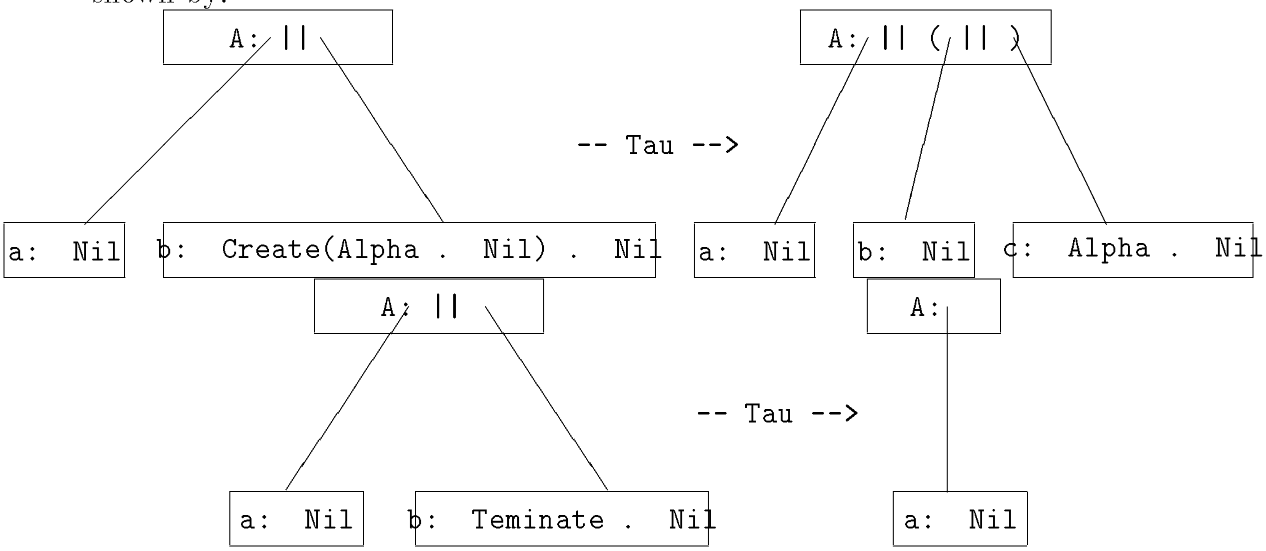
`if ep -- Terminates --> ep' then pid:(ep || pg) -- Tau --> pid: pg`

`if ep -- Creates(p) --> ep' then`

`pid:(ep || pg) -- Tau --> pid:(ep' || id: p || pg)`

for each `id` not in `pg`

Graphically an example of a creation and of a termination of a process of CD_3 are shown by:



End example

3.4.3 Entity specifications

The dynamic systems which have particular structural properties may be abstractly specified using entity specifications.

The entity specifications are particular dynamic specifications characterized by:

- entity signature;
- formulae/predicates for investigating the concurrent structure (e.g. only parallelism at one level, absence of deadlocks and so long).

Notice that for investigating the concurrent structure the formulae of the first order logic on the signature are not sufficient; an idea is that of introducing a special predicate, called *Are_Sub* , expressing the fact that a set of entities consists of all the subentities w.r.t. a view of the structure of an entity. Moreover since we need to handle sets of entities, other the predicate *Are_Sub* , we add also sorts/constants/operations/predicates of the data type sets of entities.

Thus an *entity specification* is a pair $(E\Sigma, AX)$, where AX is a set of formulae on $E\Sigma$ enriched by:

- the special predicates _ *Are_Sub* _: *set-ente-s*, for each s entity sort of $E\Sigma$, where *set-ent* is the sort of the set of the entities of all sort and whose validity on an algebra EA is defined by

  EA $\models \{e_1, e_2, \ldots, e_n\}$ *Are_Sub* e iff

  $\exists$ a view of the structure v s.t. e_1, e_2, $\ldots$, e_n are all the subentities of e w.r.t. v;
- the sort *set-ent* of the set of the entities of all sort and the constants/operations/ predicates on such finite sets of entities.

Example 3.6 Specification of concurrent systems with structural properties
Consider the problem of specifying the concurrent systems consisting of 5 simple processes (i.e. without internal parallelism) put in parallel where the deadlock situations may never arise using an entity specification.

The signature of the specification is
`SYST = esorts syst proc:  _ -- _ --> _`

Now we have to express then expressing the following properties:

1. the processes have not dynamic components (they are simple),
2. the systems have 5 dynamic components of type process,
3. if a system may not evolve, then also each of its components may not evolve (absence of deadlocks).

We can express the above properties with the following axioms, where `p`, `set`, `s` are variables respectively of sorts `proc`, `ent_set` and `syst`:

1. `if set Are_Sub p then set =` $\{$`p`$\}$ (recall that each entity is a subentity of itself)

2. `if set Are_Sub s then Card(set) = 5`

   `if set Are_Sub s and x In set then exists p :  p = x`

3. `if (not exists s', ls:  s -- ls --> s') and set Are_Sub s and p In set then`

   `(not exists p',lp:  p -- lp --> p')`

End example

References

[AMRW85] E. Astesiano, G.F. Mascari, G. Reggio, and M. Wirsing. On the parameterized algebraic specification of concurrent systems. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Proc. TAPSOFT'85, Vol. 1*, number 185 in Lecture Notes in Computer Science, pages 342–358. Springer Verlag, Berlin, 1985.

[AR87a] E. Astesiano and G. Reggio. An outline of the SMoLCS approach. In M. Venturini Zilli, editor, *Mathematical Models for the Semantics of Parallelism, Proc. Advanced School on Mathematical Models of Parallelism, Roma, 1986*, number 280 in Lecture Notes in Computer Science, pages 81–113. Springer Verlag, Berlin, 1987.

[AR87b] E. Astesiano and G. Reggio. The SMoLCS approach to the formal semantics of programming languages – A tutorial introduction. In A.N. Habermann and U. Montanari, editors, *System Development and Ada, Proc. of CRAI Workshop on Software Factories and Ada, Capri 1986*, number 275 in Lecture Notes in Computer Science, pages 81–116. Springer Verlag, Berlin, 1987.

[AR87c] E. Astesiano and G. Reggio. SMoLCS-driven concurrent calculi. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. TAPSOFT'87, Vol. 1*, number 249 in Lecture Notes in Computer Science, pages 169–201. Springer Verlag, Berlin, 1987.

[AR92] E. Astesiano and G. Reggio. A structural approach to the formal modelization and specification of concurrent systems. Technical Report PDISI-92-01, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Italy, 1992.

[CR91] G. Costa and G. Reggio. Abstract dynamic data types: a temporal logic approach. In A. Tarlecki, editor, *Proc. MFCS'91*, number 520 in Lecture Notes in Computer Science, pages 103–112. Springer Verlag, Berlin, 1991.

[PR94] F. Parodi and G. Reggio. Metal: a metalanguage for SMoLCS. Technical Report DISI–TR–94–13, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1994.

[Reg91] G. Reggio. Entities: an institution for dynamic systems. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, number 534 in Lecture Notes in Computer Science, pages 244–265. Springer Verlag, Berlin, 1991.

Contents