CASL-LTL
A CASL EXTENSION FOR DYNAMIC SYSTEMS
Summary

CoFI Document: CASL/Summary
Version: 1.0 8 August 2003

G. Reggio - E. Astesiano - C. Choppy
E-mail address for comments: reggio@disi.unige.it

COFI: The Common Framework Initiative
http://www.brics.dk/Projects/CoFI

*This document is available on WWW**, *and by FTP*†.

Abstract

CASL the basic language developed within COFI, the Common Framework Initiative for algebraic specification and development, cannot be used for specifying the requirements and the design of dynamic software systems. CASL-LTL is an extension to overcome this limit, allowing to specify dynamic system by modelling them by means of labelled transition systems and by expressing their properties with temporal formulae. It is based on LTL, the Labelled Transition Logic, that is a logic-algebraic formalism for the specification of dynamic systems, mainly developed by E.Astesiano and G. Reggio (see [AR01] and [CR97]).

This document gives a detailed summary of the syntax and intended semantics of CASL-LTL. It is intended for readers who are already familiar with CASL ([Mos03]).

Four short examples are given in the appendix, and extended case studies using CASL-LTL are given in [CR00, CR03]. An extensive companion user method is given in [CR03] (while [CR00] gives a first attempt to rely on structuring concepts). CASL-LTL was also used to present the semantics of some parts of UML in [RACH00, RCA01].

*...
†.....

Chapter 1

Casl-Ltl

[ABR99] present different ways of exploiting algebraic methods in concurrency. The CASL-LTL extension (**A3** "algebraic specifications of dynamic-data types" approach of [ABR99]) uses dynamic sorts, the elements of which correspond to concurrent systems. It is based on LTL, the Labelled Transition Logic, that is a logic-algebraic formalism for the specification of dynamic systems, mainly developed by E.Astesiano and G. Reggio (see [AR01] and [CR97]). While [CR97] explains the differences with the temporal logics developed in [MP89, MP92] (e.g., LTL is branching-time instead of linear), the connectors and formula of LTL are close to those of CTL* [Eme90], and LTL is anchored (cf. footnote 1 page 4), first-order many-sorted and provides also edge formulas.

1.1 Basic Concepts

CASL-LTL is an extension of CASL for the specification of what we call in a general way *dynamic systems*, as processes, concurrent, reactive, distributed, parallel, ... systems. The basic idea behind CASL-LTL is to model a dynamic system by using a labelled transition system.

A *labelled transition system* (shortly *lts*) is a triple $(STATE, LABEL, \rightarrow)$, where $STATE$ and $LABEL$ are two sets, and $\rightarrow \subseteq STATE \times LABEL \times STATE$ is the *transition relation*. A triple $(s, l, s') \in \rightarrow$ is said to be a *transition* and is usually written $s \xrightarrow{l} s'$.

Given an lts we can associate with each $s_0 \in STATE$ the tree (*transition tree*) whose root is s_0, where if it has a node n decorated with s and $s \xrightarrow{l} s'$, then it has a node n' decorated with s' and an arc decorated with l from

n to n', the order of the branches is not considered, and two identically decorated subtrees with the same root are considered as a unique subtree.

We model a dynamic system S with a transition tree determined by an lts

$$(STATE, LABEL, \rightarrow)$$

and an initial state $s_0 \in STATE$; the nodes in the tree represent the intermediate (interesting) situations of the life of S, and the arcs of the tree the possibilities of S of passing from one situation to another. It is important to note here that an arc (a transition) $s \xrightarrow{l} s'$ has the following meaning: S in the situation s has the *capability* of passing into the situation s' by performing a transition, where the label l represents the interaction with the environment during such a move; thus l contains information on the conditions on the environment for the capability to become effective, and on the transformation of such environment induced by the execution of the transition.

An lts may be specified by an algebraic specification having two sorts,

$$\mathit{State} \qquad \text{and} \qquad \mathit{Label},$$

whose elements are the states and the labels of the lts respectively, and a ternary predicate

$$\_\_\texttt{--}\_\_\texttt{-->}\_\_ : \mathit{State} \times \mathit{Label} \times \mathit{State}$$

corresponding to the transition relation.

The basic idea behind CASL-LTL is to add to CASL a special construct to declare such two sorts and the associate predicate that correspond to an lts; precisely the declaration of *dynamic sort*

dsort *Ds* **label** *Label_Ds*

where *Ds* is called a *dynamic sort* and *Label_Ds* is its label sort. This terminology wants to remind that the elements of *Ds* are dynamic, indeed they correspond to a dynamic system in a particular initial situation.

dsort *Ds* **label** *Label_Ds* corresponds to the following declarations

sorts *Ds*, *Label_Ds*
pred __--__-->__ : $Ds \times Label\_Ds \times Ds$.

The CASL formulae (many sorted first order logic) built by using the transition predicate (__--__-->__) allows to express some properties on the behaviour of a dynamic system, but they are not sufficient. For example, using such formulae we cannot state liveness properties as, "eventually the system will send out some value" (i.e., eventually it will perform a transition whose label correspond to send out such value). Instead such properties could be easily expressed by using some kind of temporal logic. Thus CASL-LTL includes temporal logic combinators, and precisely those of the many-sorted first-order temporal logic of [CR97]. Clearly, the temporal formulae are sensible only when referring to elements of dynamic sorts.

1.2 Dynamic Signatures

A *dynamic many-sorted signature* $D\Sigma = (S, DS, TF, PF, P)$ consists of:

- a set S of *sorts*;
- a set DS of *dynamic sorts* s.t. $DS \subseteq S$ and for all $Ds \in DS$ there exists $Label\_Ds \in S - DS$;
- sets $TF_{w,s}$, $PF_{w,s}$, of *total function symbols*, respectively *partial function symbols*, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each *function profile* (w, s) consisting of a sequence of *argument sorts* $w \in S^*$ and a *result sort* $s \in S$;
- sets P_w of *predicate* symbols, for each *predicate profile* consisting of a sequence of argument sorts $w \in S^*$, s.t. for each $Ds \in DS$ $\_\_\texttt{--}\_\_\texttt{-->}\_\_ \in P_{Ds\ Label\_Ds\ Ds}$.

A *dynamic many-sorted signature morphism*

$$\sigma : (S, DS, TF, PF, P) \to (S', DS'\, TF', PF', P')$$

consists of a mapping from S to S', and for each $w \in S^*, s \in S$, a mapping between the corresponding sets of function, resp. predicate symbols, s.t. dynamic sorts are sent into dynamic sorts, and the associated label sort and transition predicate are sent into the corresponding label sort and transition predicate. Precisely:

$\sigma(Label\_Ds) = Label\_\sigma(Ds)$ and

$\sigma(\_\_\texttt{--}\_\_\texttt{-->}\_\_ : Ds \times Label\_Ds \times Ds) = \_\_\texttt{--}\_\_\texttt{-->}\_\_ : \sigma(Ds) \times Label\_\sigma(Ds) \times \sigma(Ds)$.

1.3 Models

Assume that $D\Sigma = (S, DS, TF, PF, P)$ is a dynamic many-sorted signature.

A *dynamic many-sorted model* for $D\Sigma$ $M \in \mathbf{Mod}(D\Sigma)$ is defined as for the basic CASL.

A (weak) *dynamic many-sorted homomorphism* h from M_1 to M_2, with $M_1, M_2 \in$ $\mathbf{Mod}(D\Sigma)$ is defined as for the basic CASL.

However, the explicit presence of the dynamic sorts allows us to equip a dynamic model M with a set of "paths" for each dynamic sort Ds representing the possible behaviours of the elements of sort Ds in M.

Paths are defined precisely as follows. For each $M \in \mathbf{Mod}(D\Sigma)$ and $Ds \in DS$, the set of the *paths* on M of sort Ds, denoted by $PATH(M, Ds)$, is the set of all the sequences of transitions having the form either (1) or (2) below:

(1) $s_0\ l_0\ s_1\ l_1\ s_2\ l_2\ \ldots$ (infinite path)
(2) $s_0\ l_0\ s_1\ l_1\ s_2\ l_2\ \ldots\ s_n$ $n \geq 0$

where for all $i \geq 0$, $s_i \in Ds^M$, $l_i \in Label\_Ds^M$, $(s_i, l_i, s_{i+1}) \in \rightarrow^M$, and there do not exist l, s' such that $(s_n, l, s') \in \rightarrow^M$.

If $\sigma = s_0\ l_0\ s_1\ l_1\ s_2\ l_2\ \ldots$ and $\sigma \in PATH(M, Ds)$
- given $h \geq 0$, if there exists s_h, then $\sigma|_h$ denotes the path $s_h\ l_h\ s_{h+1}\ l_{h+1}\ s_{h+2}\ \ldots$ and is referred to as "σ at point h", otherwise it is undefined,
- *first_state*(σ) denotes s_0, the *first state* of σ, and *first_label*(σ) denotes l_0, the *first label* of σ, if exists, i.e., if σ is not just a state.

1.4 Sentences

For a dynamic many-sorted signature $D\Sigma = (S, DS, TF, PF, P)$ the *dynamic many-sorted sentences* in $\mathbf{TSen}(D\Sigma)$ are the usual closed many-sorted first-order logic formulae, built from atomic formulae using quantification (over sorted variables) and logical connectives plus temporal formulae anchored to the elements of the dynamic sorts.

The temporal formulae of CASL-LTL express properties of the elements of a dynamic sort Ds (dynamic elements) in terms of their paths, i.e., of their possible behaviours. Such temporal formulae have form

either $in\_any\_case(t, \pi)$ or $in\_one\_case(t, \pi)$

where t is a term of sort Ds, and π a path formula. The first formula can be read as "for every path σ starting in the state denoted by t, π holds on σ", while the second means "there exists a path σ starting in the state denoted by t s.t. π holds on σ".[1] The path formulae for the elements of a dynamic

[1] We anchor these formulae to states, following the ideas in [MP89]. The major difference with the classical temporal logic is that we do not specify a single system but, in general, one or many types of systems, so there is not a single initial state but several, hence the need for an explicit reference to states (through terms) in the formulae built with *in_any_case*.

sort Ds express properties on its paths, i.e., on the possible behaviours of the elements of Ds.

The dynamic sentences in $\mathbf{TSen}(D\Sigma)$ and the path formulae for the elements of the various dynamic sorts of $D\Sigma$ ($\mathbf{PSen}(D\Sigma, Ds)_{Ds \in DS}$) are defined as follows:

$\mathbf{TSen}(D\Sigma)$ contains all the atomic formulae of the basic CASL, all those built with the logic combinator of the basic CASL, and the following temporal formulae: for each $Ds \in DS$

$in\_any\_case(t, \pi)$ and $in\_one\_case(t, \pi)$

with t term of sort Ds and $\pi \in \mathbf{PSen}(D\Sigma, Ds)$.

The path formulae over $D\Sigma$ for the elements of sort Ds, $\mathbf{PSen}(D\Sigma, Ds)$, are defined as follows

- $[\ x \bullet F\ ]$ — x variable of sort Ds, $F \in \mathbf{TSen}(D\Sigma)$

  This formula holds on a path σ whenever F holds at the first state of σ.

- $<\ x \bullet F\ >$ — x variable of sort $Label\_Ds$, $F \in \mathbf{TSen}(D\Sigma)$

  This formula holds on a path σ whenever σ is not just a single state and F holds at the first label of σ.

- π_1 *until* π_2 — $\pi_1, \pi_2 \in \mathbf{PSen}(D\Sigma, Ds)$

  This formula holds on a path σ whenever there exists a point in σ s.t. π_2 holds at such point and π_1 holds until before it.

- *next* π — $\pi \in \mathbf{PSen}(D\Sigma, Ds)$

  This formula holds on a path σ whenever π holds at σ at point 1 if it exists or whenever σ at point 1 does not exist.

- *eventually* π — $\pi \in \mathbf{PSen}(D\Sigma, Ds)$

  This formula holds on a path σ whenever there exists a point in σ s.t. π holds at such point.

- *always* π — $\pi \in \mathbf{PSen}(D\Sigma, Ds)$

  This formula holds on a path σ whenever π holds at any point in σ.

- $\neg\, \pi$, $\pi \Rightarrow \pi'$, $\forall\, y \bullet \pi$

  y variable of whatever sort, $\pi, \pi' \in \mathbf{PSen}(D\Sigma, Ds)$

  with the usual meaning

1.5 Satisfaction

Let M be a dynamic model over $D\Sigma$ and v a variable evaluation, then we define by multiple induction:

- the validity of a dynamic formula $F \in \mathbf{TSen}(D\Sigma)$ in M w.r.t. v (written $M, v \models F$),
- the validity of a path formula $\pi \in \mathbf{PSen}(D\Sigma, Ds)$ on a path $\sigma \in PATH(D\Sigma, Ds)$ in M w.r.t. v (written $M, v, \sigma \models \pi$),

as follows:

- $M, v \models in\_any\_case(t, \pi)$ iff for each $\sigma \in PATH(M, Ds)$ such that $first\_state(\sigma) = t^{M,v}$, $M, v, \sigma \models \pi$
- $M, v \models in\_one\_case(t, \pi)$ iff there exists $\sigma \in PATH(M, Ds)$ such that $first\_state(\sigma) = t^{M,v}$ and $M, v, \sigma \models \pi$
- $M, v, \sigma \models [\ x \bullet F\ ]$ iff $M, v[first\_state(\sigma)/x] \models F$
- $M, v, \sigma \models <\ x \bullet F\ >$ iff

  $first\_label(\sigma)$ is defined and $M, v[first\_label(\sigma)/x] \models F$
- $M, v, \sigma \models \pi_1\ until\ \pi_2$ iff there exists $j \geq 0$ such that for all h, $0 < h < j$, $M, v, \sigma|_h \models \pi_1$ and $M, v, \sigma|_j \models \pi_2$
- $M, v, \sigma \models next\ \pi$ iff $\sigma|_1$ undefined or $\sigma|_1$ defined and $M, v, \sigma|_1 \models \pi$
- $M, v, \sigma \models eventually\ \pi$ iff there exists $j \geq 0$ such that $M, v, \sigma|_j \models \pi$
- $M, v, \sigma \models always\ \pi$ iff for all $j \geq 0$ such that $\sigma|_j$ is defined, $M, v, \sigma|_j \models \pi$
- $\neg\ F$, $F\ \Rightarrow\ F'$, $\forall\ x \bullet F$, $\neg\ \pi$, $\pi\ \Rightarrow\ \pi'$, $\forall\ x \bullet \pi$ as usual.

1.6 Basic Constructs

This section indicates the abstract and concrete syntax of the new constructs introduced by CASL-LTL to the basic specifications, and describes their intended interpretation.

1.6.1 Dynamic Sort

```
SORT-ITEM       ::= dsort-item SORT-ITEM SORT-ITEM
```

A dynamic sort declaration is written:

dsort *Ds* **label** *Label_Ds*

and implicitely corresponds to the following declarations

sorts *Ds*, *Label_Ds*
pred `__--__-->__` : $Ds \times Label\_Ds \times Ds$.

1.7 Axioms

```
FORMULA          ::= TEMPORAL
```

A Casl-Ltl formula may be also a temporal formula.

1.7.1 Temporal Formulae

```
TEMPORAL            ::= temporal PATH-QUANTIFIER TERM PATH-FORMULA
PATH-QUANTIFIER     ::= anycase | onecase
```

A temporal formula with the `anycase` quantifier is written:

$in\_any\_case(T, PF)$

A temporal formula with the `onecase` quantifier is written:

$in\_one\_case(T, PF)$

The first case is the universal path quantification, holding when PF holds for all paths starting from the element represented by the term T; the second case is the existential path quantification, holding when PF holds for at least one path starting from the element represented by T.

1.7.2 Path Formulae

These formulae represent properties on the paths, i.e., on the possible behaviours of the dynamic elements.

```
PATH-FORMULA   ::= STATE-COND | LABEL-COND |
                   UNTIL | NEXT | EVENTUALLY | ALWAYS |
                   P-QUANTIFICATION |
                   P-CONJUNCTION | P-DISJUNCTION |
                   P-IMPLICATION | P-EQUIVALENCE | P-NEGATION
```

1.7.2.1 State Condition

```
STATE-COND   ::= state-cond VAR-DECL FORMULA
```

A state condition is written:

$[\ VD \bullet F\ ]$

1.7.2.2 Label Condition

```
LABEL-COND   ::= label-cond VAR-DECL FORMULA
```

A label condition is written:

$<\ VD \bullet F\ >$

1.7.2.3 Until

```
UNTIL ::= until PATH-FORMULA PATH-FORMULA
```

An until formula is written:

PF_1 *until* PF_2

1.7.2.4 Next

```
NEXT ::= next PATH-FORMULA
```

A next formula is written:

next PF

1.7.2.5 Eventually

```
EVENTUALLY ::= eventually PATH-FORMULA
```

An eventually formula is written:

eventually PF

1.7.2.6 Always

```
ALWAYS ::= always PATH-FORMULA
```

An always formula is written:

always PF

1.7.2.7 First order combinators on path formulae

```
P-QUANTIFICATION   ::= pquantification QUANTIFIER VAR-DECL+ PATH-FORMULA
P-CONJUNCTION      ::= pconjunction PATH-FORMULA+
P-DISJUNCTION      ::= pdisjunction PATH-FORMULA+
P-IMPLICATION      ::= pimplication PATH-FORMULA PATH-FORMULA
P-EQUIVALENCE      ::= pequivalence PATH-FORMULA PATH-FORMULA
P-NEGATION         ::= pnegation PATH-FORMULA
```

These formulae are written as the corresponding one of the basic CASL.

1.8 Structured Specifications

The structuring constructs of CASL-LTL are exactly the same of the basic CASL, clearly defined using the new signature morphisms, which preserve dynamicity of sorts, and the associated labels sorts and transition predicates (see [CR97] for the precise definitions).

1.9 Architectural Specifications

The relationship of the new constructs of CASL-LTL with the architectural specifications need further investigations.

1.10 Specification Libraries

As for the basic CASL.

Bibliography

[ABR99] E. Astesiano, M. Broy, and G. Reggio. Algebraic Specification of Concurrent Systems. In E. Astesiano, B. Krieg-Brückner, and H.-J. Kreowski, editors, *IFIP WG 1.3 Book on Algebraic Foundations of System Specification*. Springer Verlag, 1999.

[AR01] E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. *Acta Informatica*, 37(11-12), 2001.

[CR97] G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2), 1997.

[CR00] C. Choppy and G. Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 14th International Workshop WADT'99*, number 1827 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000. A complete version is available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.

[CR03] C. Choppy and G. Reggio. Towards a Formally Grounded Software Development Method. Technical Report DISI–TR–03–35, DISI, Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio03a.pdf`.

[Eme90] A.E. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B. Elsevier, 1990.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.

[Mos03] P.D. Mosses, editor. *CASL, The Common Algebraic Specification Language - Reference Manual*. Lecture Notes in Computer Science. Springer-Verlag, 2003. To appear. Available at `http://www.cofi.info/CASL_RefManual_DRAFT.pdf`.

[MP89] Z. Manna and A. Pnueli. The Anchored Version of the Temporal Framework. In J.W. de Bakker, W.-P. de Roever, and G. Rozemberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 354 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logics of Reactive and Concurrent Systems*. Springer Verlag, New York, 1992.

[RACH00] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000*, number 1783 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.

[RCA01] G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In H. Hussmann, editor, *Proc. FASE 2001*, number 2029 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.

.1 Abstract Syntax

The abstract syntax of CASL-LTL is given by extending with the following clauses the part concerning basic specifications of that the basic CASL (see [Mos03]).

```
SORT-ITEM          ::= dsort-item SORT-ITEM SORT-ITEM

FORMULA            ::= TEMPORAL

TEMPORAL           ::= temporal PATH-QUANTIFIER TERM PATH-FORMULA
PATH-QUANTIFIER    ::= anycase | onecase

PATH-FORMULA       ::= STATE-COND | LABEL-COND |
                       UNTIL | NEXT | EVENTUALLY | ALWAYS |
                       P-QUANTIFICATION | P-CONJUNCTION | P-DISJUNCTION |
                       P-IMPLICATION | P-EQUIVALENCE | P-NEGATION

STATE-COND         ::= state-cond VAR-DECL FORMULA
LABEL-COND         ::= label-cond VAR-DECL FORMULA
UNTIL              ::= until PATH-FORMULA PATH-FORMULA
NEXT               ::= next PATH-FORMULA
EVENTUALLY         ::= eventually PATH-FORMULA
ALWAYS             ::= always PATH-FORMULA
P-QUANTIFICATION   ::= pquantification QUANTIFIER VAR-DECL+ PATH-FORMULA
P-CONJUNCTION      ::= pconjunction PATH-FORMULA+
P-DISJUNCTION      ::= pdisjunction PATH-FORMULA+
P-IMPLICATION      ::= pimplication PATH-FORMULA PATH-FORMULA
P-EQUIVALENCE      ::= pequivalence PATH-FORMULA PATH-FORMULA
P-NEGATION         ::= pnegation PATH-FORMULA
```

.2 Abbreviated Abstract Syntax

The abbreviated abstract syntax of CASL-LTL is given by extending with the following clauses the part concerning basic and subsorted specifications of that the basic CASL (see [Mos03]).

```
SIG-ITEM          ::= dsort-item SORT-ITEM SORT-ITEM

FORMULA           ::= temporal PATH-QUANTIFIER TERM PATH-FORMULA

PATH-QUANTIFIER   ::= anycase | onecase

PATH-FORMULA      ::= state-cond VAR-DECL FORMULA |
                      label-cond VAR-DECL FORMULA |
                      until PATH-FORMULA PATH-FORMULA |
                      next PATH-FORMULA |
                      eventually PATH-FORMULA |
```

```
always PATH-FORMULA |
pquantification QUANTIFIER VAR-DECL+ PATH-FORMULA |
pconjunction PATH-FORMULA+ |
pdisjunction PATH-FORMULA+ |
pimplication PATH-FORMULA PATH-FORMULA |
pequivalence PATH-FORMULA PATH-FORMULA |
pnegation PATH-FORMULA
```

.3 Concrete Syntax

The concrete syntax of CASL-LTL is given by extending with the following clauses the part concerning basic specifications with subsorts of that the basic CASL (see [Mos03]).

```
SIG-ITEM         ::= dsort SORT-ITEM label SORT-ITEM ;

FORMULA          ::= PATH-QUANTIFIER "(" TERM "."  PATH-FORMULA ")"

PATH-QUANTIFIER ::= in_any_case | in_one_case

PATH-FORMULA     ::= "[" VAR-DECL "." PATH-FORMULA "]"
                   | "<" VAR-DECL "." PATH-FORMULA ">"
                   | PATH-FORMULA  until  PATH-FORMULA
                   | next PATH-FORMULA
                   | eventually PATH-FORMULA
                   | always PATH-FORMULA
                   | QUANTIFIER  VAR-DECL ;...; VAR-DECL "." PATH-FORMULA
                   | PATH-FORMULA /\ PATH-FORMULA /\.../\ PATH-FORMULA
                   | PATH-FORMULA \/ PATH-FORMULA \/...\/ PATH-FORMULA
                   | PATH-FORMULA => PATH-FORMULA
                   | PATH-FORMULA if PATH-FORMULA
                   | PATH-FORMULA <=> PATH-FORMULA
                   | not PATH-FORMULA
                   | true | false
```

.4 Disambiguation

The context-free grammar given in Section .3 for input syntax is quite ambiguous. This section explains various precedence rules for disambiguation, and the intended grouping of mixfix formulae and terms (which are to be recognized in a separate phrase, dependent on the declared symbols and parsing annotations).

Within a `FORMULA`, the use of the new path quantifiers (`in_any_case` and `in_one_case`) do not cause any problem, due to the fact that they have a "functional syntax".

Within a `PATH-FORMULA`, the use of prefix and infix notation for the logical connectives gives rise to some potential ambiguities. These are resolved as follows:

- the state and label condition combinators (`[ . ]` and `< . >`) do not cause any problem, due to the fact that have a "functional syntax".
- The first-order logical connectives have a precedence higher than any temporal combinator, and their relative precedences are as in basic CASL (see [Mos03]).

  '`PATH-FORMULA until PATH-FORMULA`' has the highest precedence; and when repeated, '`until`' groups to the right;
- '`always PATH-FORMULA`' has lower precedence;
- '`eventually PATH-FORMULA`' has even lower precedence.

For what concerns the implicit mix-fix predicates associated with dynamic sorts (`__ -- __ --> __`) have the lowest precedence.

.5 Lexical Syntax

The lexical syntax of CASL-LTL is as for the basic CASL, except that `NO-BRACKET-SIGNS` cannot be also one of the following *reserved symbols*:

```
-- --> < > [ ]
```

.6 Display Format

The input symbols introduced in CASL-LTL in the following table are to be displayed as the mathematical symbols shown below them.

in_any_case	in_one_case	eventually	always	until	next	`__ -- __ --> __`
$\triangle$	$\triangledown$	$\diamond$	$\square$	$\mathcal{U}$	$\circ$	$\_\_ \xrightarrow{\_\_} \_\_$

There exists also another possibility more text oriented, where we have only

`__ -- __ --> __`
$\_\_ \xrightarrow{\_\_} \_\_$

.7 Example 1: CCS

spec CHANNEL =
  free type *Channel* ::= α | β | γ | η | ...

spec CCS =
  CHANNEL **then**
free {
  types *Behaviour* ::= *nil* | __.__(*Label_Behaviour*; *Behaviour*) |
  __ + __(*Behaviour*; *Behaviour*) | __||__(*Behaviour*; *Behaviour*);
  Label_Behaviour ::= __!(*Channel*) | __?(*Channel*) | τ

  dsort *Behaviour* **label** *Label_Behaviour*
  ops __ + __ : *Behaviour* × *Behaviour* → *Behaviour* **assoc**; **comm**; **idem**; **unit**: *nil*
  __||__ : *Behaviour* × *Behaviour* → *Behaviour* **assoc**; **comm**;
  vars B, B_1, B_2, B'_1, B'_2 : *Behaviour*; C : *Channel*; L : *Label_Behaviour*;

- $L.B \xrightarrow{L} B$
- $B_1 \xrightarrow{L} B'_1 \;\Rightarrow\; B_1 + B_2 \xrightarrow{L} B'_1$
- $B_1 \xrightarrow{L} B'_1 \;\Rightarrow\; B_1||B_2 \xrightarrow{L} B'_1||B_2$
- $B_1 \xrightarrow{C\,!} B'_1 \;\wedge\; B_2 \xrightarrow{C\,?} B'_2 \;\Rightarrow\; B_1||B_2 \xrightarrow{\tau} B'_1||B'_2$

} **end**

Note that, given the properties (associativity, commutativity, ...) declared for the __ + __ and __||__ operations, fewer axioms are needed to describe the transition relation than in the original CCS description [Mil80].

.8 Example 2: Fancy processes

spec PROC =
  INT **then**
  dsort *Proc* **label** *Label_Proc*
  preds *rec?*, *send?* : *Int* × *Label_Proc*
  %% checks whether a label corresponds to receive/send an integer
  vars *P*, *P'* : *Proc*; *I* : *Int*; *L*, *Y* : *Label_Proc*

- $P \xrightarrow{L} P' \;\wedge\; (\neg\, I = 0) \;\wedge\; rec?(L, I) \;\Rightarrow$
  $in\_any\_case(P', eventually\ (<\ Y \bullet send?(I+1, Y)\ > \wedge$
  $eventually\ <\ Y \bullet send?(I+2, Y)\ >))$

  %% if a process receives a non-zero integer *I, then in any case eventually*
  %% (it will send out *I+1 and eventually also I+2)*
- $P \xrightarrow{L} P' \;\wedge\; rec?(L, 0) \;\Rightarrow$
  $in\_any\_case(P', always\ <\ Y \bullet (\neg\, send?(0, Y)) \;\wedge\; (\neg\, rec?(0, Y))\ >)$

  %% if a process receive 0, then in any case forever it will never send out or receive 0

.9 Example 3: A Fancy Concurrent System

spec CSystem =
    Proc **then**
free {
    types $System ::= \emptyset \mid \_\_||\_\_(Proc; System)$
    $Label\_System ::= I(Int) \mid O(Int) \mid \tau$
    dsort *System* **label** *Label_System*
}
    vars $P, P' : Proc; I : Int; Y : Label\_Proc; S : System; Z : Label\_System$
- $P \,||P' \,||S = P' \,||P \,||S$
- $in\_one\_case(P, eventually < Y \bullet send?(I, Y) >) \Rightarrow$
  $in\_one\_case(P \,||S, eventually < Z \bullet Z = O(I) >)$

    %% if a process component of the system has at least in a case the capability
    %% to eventually send out *I, then also the system has such capability*

.10 Example 4: A Buffer (The Bit example)

This example is a very simple concurrent system consisting of a buffer and a user. It is takem from [ABR99] and its specification is now expressed in Casl-Ltl.

The system **Bit** (called Bit since it is really very small) consists of two components in parallel: a user and a buffer. The buffer is organized as a queue and contains integers; it may obviously receive and return integer values; it may break down, in which case its content will be 10^{10}, and, moreover, it may happen that the last element of its content is duplicated.

When the system is started by the environment, the buffer is empty and the user puts in sequence 0 and 1 on the buffer; then it gets the first element from the buffer. If this element is the number *0* the user must inform the environment of the correct working of the buffer, otherwise it must signal that there is an error.

Thus **Bit** is an interactive concurrent system with components having both autonomous activities (as the buffer failures) and cooperations (the user writing/reading the buffer), and using some static data (integers); furthermore it also has some relevant static/functional aspects, as the queue organization of the buffer.

Some relevant requirements on **Bit** are:

R0 The buffer must always be able to receive any integer value.

R1 When the user is terminated, it cannot perform an activity again.

R2 In at least one case, the system must behave correctly.

R3 After being started, it will eventually signal *ok* or *error*

R4 *ok* and *error* are signaled at most once, and it cannot happen that both are signaled.

R5 The user puts integers on and gets integers from the buffer.

We first specify the two components of the system, the buffer and the user, and then how they cooperate.

spec BUFFER =
INT_QUEUE **with sort** *Queue* $\mapsto$ *Buffer* **then**
dsort *Buffer* **label** *Lab_Buffer*
ops τ :$\rightarrow$ *Lab_Buffer*
receive, *return* : *Int* $\rightarrow$ *Lab_Buffer*
vars B : *Buffer*; I : *Int*

- $not\_empty(B) \Rightarrow B \xrightarrow{return(first(B))} remove(B)$
- $B \xrightarrow{receive(I)} put(I, B)$
- $B \xrightarrow{\tau} put(10^{10}, empty)$
- $not\_empty(B) \Rightarrow B \xrightarrow{\tau} dup(B)$

spec USER_STATUS =
INT **then**
sort *User_Status*
ops *initial*, *putting_0*, *putting_1*, *reading*, *terminated* :$\rightarrow$ *User_Status*
read : *Int* $\rightarrow$ *User_Status*

spec USER =
USER_STATUS **with sort** *User_Status* $\mapsto$ *User* **then**
dsort *User* **label** *lab_User*
ops *start*, *ok*, *error* :$\rightarrow$ *lab_User*
put, *get* : *Int* $\rightarrow$ *lab_User*
vars I : *Int*

- $initial \xrightarrow{start} putting\_0$
- $putting\_0 \xrightarrow{put(0)} putting\_1$
- $putting\_1 \xrightarrow{put(1)} reading$
- $reading \xrightarrow{get(I)} read(I)$
- $read(0) \xrightarrow{ok} terminated$
- $not\_eq(I, 0) \Rightarrow read(I) \xrightarrow{error} terminated$

spec SYSTEM =
BUFFER **and** USER **then**
dsort *System* **label** *Lab_System*
ops _ | _ : *Buffer* $\times$ *User* $\rightarrow$ *System*
start, *ok*, *error*, τ :$\rightarrow$ *Lab_System*
vars B, B': *Buffer*; U, U': *User*

- $U \xrightarrow{start} U' \Rightarrow empty \mid U \xrightarrow{start} empty \mid U'$
- $B \xrightarrow{receive(I)} B' \wedge U \xrightarrow{put(I)} U' \Rightarrow B \mid U \xrightarrow{\tau} B' \mid U'$

- $B \xrightarrow{return(I)} B' \wedge\ U \xrightarrow{get(I)} U' \Rightarrow b \mid U \xrightarrow{\tau} B' \mid U'$
- $U \xrightarrow{ok} U' \Rightarrow B \mid U \xrightarrow{ok} b \mid U'$
- $U \xrightarrow{error} U' \Rightarrow B \mid U \xrightarrow{error} B \mid U'$
- $B \xrightarrow{\tau} B' \Rightarrow B \mid U \xrightarrow{\tau} B' \mid U$