ILDI : interpreter of LOTOS ADTs

Tomá Robles Valladares

Dpt. Ingeniería Telemática
E.T.S.I. de Telecomunicación
Univ. Politécnica de Madrid
E-28040 Madrid, SPAIN

<lotos@dit.upm.es>

December 17, 1992

Abstract

ILDI is an interactive interpreter of LOTOS ADTs. It allows the testing and debugging of LOTOS ADTs. ILDI uses $\Lambda\delta$-machine format as input. This format is fully compatible with those used by the idle (compiler of ADTs used by TOPO).

The interface is provided through an interactive command menu, and may also be used in bath mode.

1 Introduction

The LOTOS compiler TOPO incorporates a Compiler of ADTs. This compiler generated a *C* or *Ada* code that is compiled with others pieces of code and provides the implementation of the LOTOS Specification.

The testing and debugging of LOTOS ADTs requires a compilation process. In this environment ILDI objectives are:

- To provide a useful tool for debugging and testing LOTOS ADTs.
- Easiness of use, both in interactive or in bath mode.
- To avoid the compilation phase.
- Full compatibility with TOPO compiler.

As in the TOPO compiler LOTOS equations are interpreted as rewriting rules, and equality is based on comparing the canonical forms.

2 ILDI structure

In order to complete the previous objectives, ILDI has the next characteristics:

2.1 Input: spec.ldi

The LOTOS specification has to be processed using: lfe (syntactic checker), lsa (semantics checker), and idle (intermediate code generator). It is the same file used as input for the LOTOS compiler.

The input format is described in [**?**].

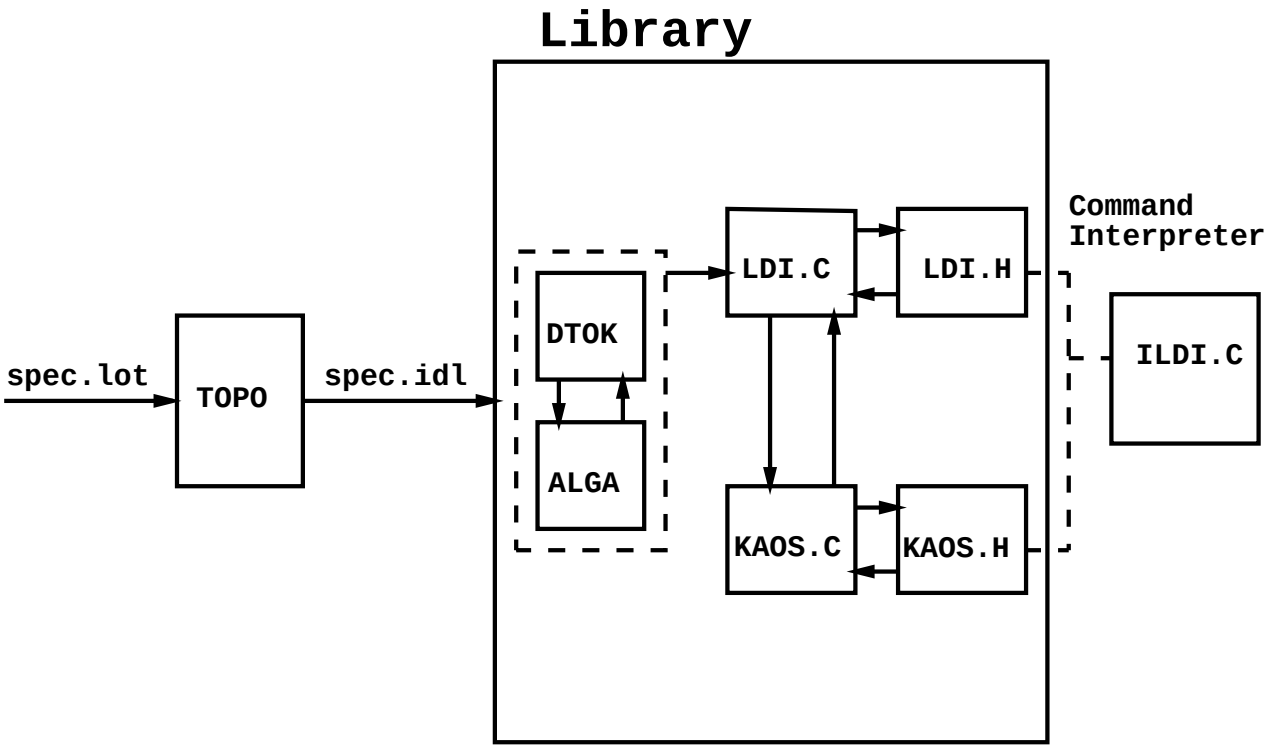

Figure 1: General Diagram

2.2 Architecture

The internal architecture of ILDI is divided into two main parts:

Library interface: The core of ildi is formed by 4 files that offer a library of functions to perform: equality resolution and rewriting of expressions. Those files are:

ldi.h and ildi.c: They generate the structures used in the interpretation of expressions. These structures are built when reading the file spec.ldi. Ildi files are generated for each specification using DTOK and ALGA modules.

The structures generated are:

- Sort declaration list.
- Operation declaration list.
- Equations for rewriting.

kaos.h and kaos.c: They implement the operational part of the interpreter. They provide functions that implement the rewriting and equality resolution. It also offers a function for drawing expressions.

They are general for whatever specification, and it is not necessary to generate them each time.

Command interpreter: The interface of the ildi is offered by a command interpreter. It works like a shell that reads commands and makes the corresponding calls to functions of kaos or ildi.

3 Manual

NAME

ildi: interactive interpreter of LOTOS ADTs.

SYNOPSIS

ildi.

DESCRIPTION

"ildi" is a tool for testing and debugging LOTOS ADTs. It allows the rewriting of expressions and the checking of equality, inside the environment of a LOTOS Specification.

"ildi" ildi reads an specification previously processed with TOPO. To get the appropriate file, it performs the next call:

```
topo spec -ildi
```

COMMANDS

"ildi" offers a menu with the next commands:

- `load <spec>`

  It loads in the environment of work the specification of file <spec>.ldi. Each time this command is executed, every previous information is lost.

- `rewrite <expression> [<sort>]`

  It performs the rewriting of `<expression>` applying the rewriting rules of the specification currently loaded. If some error happens, it will be reported on the screen.

  The notation is prefix, and the arguments must be enclosed with brackets. Infix notation is ignored. The field $< sort >$ is optional and must be used only when there exist two operations with the same arguments, but different result. If it is used, the interpreter will resolve the overloading.

- `check <exp1> = <exp2> [<sort>]`

  It checks if the result of rewriting the two expressions is the same. The format of the expressions as the one described for *rewrite* command. Also the field $< sort >$ is optional, and used for overloading resolution.

- `setdeb [yes] [no]`

  It changes the trade mode of the interpreter. With the trace mode **on**, the rewrite rules applied , when rewritten an expression, are reported on the screen. If trace mode is **off** it works silently.

  OPTIONS

  - yes: it puts trace mode on.
  - no: it puts trace mode off.
  - no options: it shows the current mode.

- `display [sorts] [operations]`

  It shows on the screen the list of sorts and/or operations of the current specification.

  OPTIONS

– sorts: it shows the list of sorts.

– operations: it shows the list of operations

– no options: it shows both sorts and operations.

- `help`
  It shows on the screen the available commands with their correct syntax.

- `quit`
  It leaves the interpreter.

FILES

<spec>.lot LOTOS specification.

<spec>.ldi Specification in the input format of the interpreter.

MESSAGES

The errors detected in the loaded specification, and in the rewriting phase generate the suitable report on the screen.

It is not possible to use specifications with: the same name, the same arguments and the same result. Any other overloading is permitted, and the interpreter will solve it.

When parsing an user expression, any operation without arguments that does not match with a LOTOS constant will be interpreted as a new constant. Then it is important to take care of misspelling LOTOS constants when writing expressions.

4 Example

In this section we are going to see an example of the application of the interpreter. We are going to use a LOTOS specification with the natural number modulus four:

```
TYPE Nat14 IS
  SORTS
    boolean, Nat14
  OPNS
    true, false: -> boolean
    n1, n2, n3, n4: -> Nat14
    succ, pred: Nat14 -> Nat14
    _ plus _: Nat14, Nat14 -> Nat14
    _ eq _: Nat14, Nat14 -> boolean
  EQNS
    OFSORT Nat14
    FORALL m, n: Nat14
      n2 = succ (n1) ;
      n3 = succ (succ (n1)) ;
      n4 = succ (n3) ;
      pred (m) = succ (succ (succ (m))) ;
      m plus succ (n) = succ (m plus n) ;
      m plus n1 = succ (m) ;
      succ (succ (succ (succ (n1)))) = n1 ;
      succ (succ (succ (succ (succ (m))))) = succ (m) ;

    OFSORT boolean
    FORALL x, y: Nat14
      x eq x = true ;
      x eq succ (x) = false ;
      x eq succ (succ (x)) = false ;
      x eq succ (succ (succ (x))) = false ;
      x eq succ (succ (succ (succ (x)))) = true ;
      n1 eq succ (succ (succ (succ (y)))) = n1 eq y ;
      succ (x) eq y = x eq succ (succ (succ (y))) ;
  ENDTYPE
```

ILDI is started from `topo`:

```
...> topo nat14 -ildi
Interactive Lotos Data Interpreter ...
lfe  nat14.lot > ./tmp_topo575
mv ./tmp_topo575 nat14.lfe
lsa -f -C  < nat14.lfe > ./tmp_topo575
mv ./tmp_topo575 nat14.lsf
idle -s -r  -tnat14.at < nat14.lsf > ./tmp_topo575
mv ./tmp_topo575 nat14.ldi
ildi nat14

Lotos Data Command Interpreter.
Introduce command:
ildi>
```

Now the specification is ready to be processed. When we are working with several specifications or we do not know well the specification, it is interesting to examine the sorts and operations of the specification:

```
ildi> display sort

SORTS:
* boolean
* Nat14

ildi> display opn

OPERATIONS:
* true      -> boolean
* false      -> boolean
* n1      -> Nat14
* n2      -> Nat14
* n3      -> Nat14
* n4      -> Nat14
* succ     Nat14 -> Nat14
* pred     Nat14 -> Nat14
* plus     Nat14 , Nat14 -> Nat14
* eq     Nat14 , Nat14 -> boolean
```

Next we show some examples of rewriting and equality checking.

```
ildi> rewrite plus(succ(n1),pred(n4))
------>>>>>>     n1

ildi> check n1 = pred(plus(n1,n3))
[ n1 = pred(plus(n1,n3)) ] => FAILS

ildi> check n1 = plus(succ(n1),pred(n4))
[ n1 = plus(succ(n1),pred(n4)) ] => OK
```

To know what equations are applied to rewrite an expression, we have to use the *setdeb* command. Let's see an example of trace activation, and the messages produced when a rewriting is performed:

```
ildi> setdeb yes
Operation OK.

ildi> rewrite plus(succ(n1),pred(n4))
  pred(m) = succ(succ(succ(m))) ;
  succ(succ(succ(succ(n1)))) = n1 ;
  m plus succ(n) = succ(m plus n) ;
  m plus succ(n) = succ(m plus n) ;
  m plus n1 = succ(m) ;
  succ(succ(succ(succ(n1)))) = n1 ;
------>>>>>>     n1
```

References