Introduction of specification modules

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

$Revision: 1.5 $

Table of Contents

Eli provides a library of specification modules. Each module contains a complete set of specifications that solve a single common task in language implementation. If such a task is identified in the design of an application specification, the module is simply added to the application specification and used there. Most modules solve a rather small task. They can be flexibly combined to solve more complex problems.

The first section introduces an example that is referred to in the other sections to explain the use of modules in the context of a language implementation task. The second section explains how modules are obtained from the library and are related to users' specifications. The remaining sections describe sublibraries for a certain problem class each.

The section Section "Name Analysis Library" in *Name Analysis Library*, is a quick reference that helps to migrate specifications using modules of previous versions of this library.

This document may be used according to different strategies: In order to become familiar with problem solving by using library modules one can start with the running example of section Chapter 1 [Example], page 3, and extend it as described in the subsequent chapters. One can also start with a particular problem in mind and then search the corresponding chapter for a module that could be used to solve it. The document can also be used as a reference manual for the library modules. In any case it is recommended to read the introduction of a chapter before using one of its modules.

The modules solve their task by providing specification fragments to one or more Eli tools: Some are simply a C module, others contain `.lido`, `.pdl`, `.ptg`, or references to other library modules. The modules are designed such that as few assumptions as possible have to hold in the user's part of the specification where they are applied. E.g. `.lido` fragments supplied by library modules are mapped to the particular grammar of the user's specification by the inheritance construct of LIDO. Modules described in the Section "Abstract Data Types" in *Abstract data types to be used in specifications*, section may also be used directly in C modules or in stand-alone C programs.

Most modules in the library are generic, i.e. the names used in a module can be modified by its instantiation. This facility allows to use the same module for different purposes, e.g. a counter for variables and another for procedures, or specify the element type of a generic list type implementation. Instantiations also control the combination of modules, e.g. association of properties to procedures and to variables which might have different name spaces.

This document describes the task that each module solves, its interface, and how it can be applied in a user's specification. The descriptions are illustrated using a running example introduced in the next section (see Chapter 1 [Example], page 3). Complete executable specifications for that running example are available in the directories `$/Name/Examples` and in `$/Type/Examples`. In general that information should be sufficient to use the modules. A look at the module's text may help in special cases.

Users are encouraged to apply this design technique of decomposition into specification modules to their application as well. The structure of complex designs and their maintainability can thus be improved. The resulting modules may be reused in other applications. The techniques needed to instantiate user developed modules are described in Chapter 2 [Instantiation], page 5.

This library will be extended by adding further modules that solve common tasks. Hence, users are encouraged to contact their Eli distributor if a module for a task that might be of general interest is missing, or if a solution for such a task can be contributed.

1 Running Example

In this section we introduce an example for a language implementation task. It is used in the descriptions of the library modules of this document wherever possible. It shall serve to demonstrate the use of the modules in the context of a complete specification. Of course, such examples can only show certain aspects of the modules. They can not demonstrate their complete functionality and the variety and flexibility of possible applications.

Although the example is taken from the area of programming language implementation, similar tasks are to be solved if languages for other purposes than programming, or programming languages with other characteristics are implemented.

Complete executable specifications for this running example are available in the directories `$/Name/Examples` and in `$/Type/Examples`.

For the purpose of this example we assume that a small artificial programming language is to be implemented. The basic constructs of the language are nested blocks, declarations of variables and of names for values, assignment statements and expressions as specified by the following concrete grammar.

The overall program structure is given by

```
Program:        Source.
Source:         Block.
Block:          Compound.
Compound:       'begin' Declaration* Statement* 'end'.
```

Variable declarations specify the types of the declared variables:

```
Declaration:    'var' ObjDecls ';'.
ObjDecls:       ObjDecl // ','.
ObjDecl:        TypeDenoter Ident.
TypeDenoter:    Ident.
```

For the start of this example there is only a set of predefined type identifiers. In the type analysis section the language is extended by introduction of further types and denoters for them.

For the time being there are only three forms of statements:

```
Statement:      Expression ';'.
Statement:      Variable '=' Expression ';'
Statement:      Block.
```

Further statements are added when necessary to explain aspects of module use.

The expression syntax is left incomplete here in order to introduce operators of different precedence levels (see Section "Operator Identification" in *Type analysis tasks*). If one wants to complete this grammar without adding operators one should add the production

```
Expression:     Operand.
```

The basic operands are

```
Operand:        IntNumber.
Operand:        RealNumber.
Operand:        Variable.
Variable:       Ident.
```

The non-literal tokens are defined by

```
Ident:          PASCAL_IDENTIFIER
IntNumber:      PASCAL_INTEGER
RealNumber:     PASCAL_REAL
                PASCAL_COMMENT
```

The above grammar is chosen such that the solution of language implementation tasks of name analysis, type analysis, and translation by using library modules can be demonstrated. It is also prepared to demonstrate the combination of different module uses. The grammar is extended where necessary in the examples of subsequent sections.

2 Instantiation and Use of Modules

To use a specification module, e.g. the `AlgScope` module in the `Name` library, it is instantiated by the line

```
$/Name/AlgScope.gnrc :inst
```

in a `.specs` file. All component specification files of the module are thus included in the set of specifications. A derivation

```
x.specs:allspecs
```

can be used to inspect all the instantiated files.

The module instance obtained by the instantiation command above provides `.lido` symbol computations for a symbol named `IdUseEnv`, among other specifications.

The above instantiation command is sufficient if only one instantiation of the `AlgScope` module is used. Several instantiations of the same module are distinguished by generic instance names supplied as arguments of the instantiation

```
$/Name/AlgScope.gnrc +instance=CtrlVar :inst
```

In this case another instance of the `AlgScope` module is generated having the instance name `CtrlVar`. It may coexist with the unnamed instance created above. The names of its files and of specified entities (symbols, attributes, etc.) are prefixed by the instance identifier, e.g. `CtrlVarIdUseEnv`.

Some modules have a second generic parameter `referto`. It may specialize the module in a second dimension on instantiation: The `AlgScope` module provides compuations for `Key` attributes in `IdUseEnv` contexts that represent applied occurrences of identifiers. In some situations it may be necessary to compute more than one `Key` attribute in an `IdUseEnv` context (if the identifier is bound in different name spaces). Hence, the `referto` of the `AlgScope` module modifies the names of the `Key` attributes. If the `AlgScope` module is instantiated by

```
$/Name/AlgScope.gnrc +instance=CtrlVar +referto=Ctrl :inst
```

it provides computations for the attribute `CtrlVarIdUseEnv.CtrlKey`, among other computations.

Other modules use the `referto` parameter for different purposes, e.g. specifying the element type for a generic stack module.

If any of the two generic parameters `instance` or `referto` is omitted, as in the first two examples, their value is assumed to be the empty string.

If a module is instantiated as described its facilities can be used in certain components of the user's specification: Symbol computations as those provided by the `AlgScope` module are associated to symbols of the user's tree grammar by `.lido` constructs like

```
SYMBOL UseIdent INHERITS IdUseEnv END;
```

Note: The symbols provided by modules are `CLASS` symbols in the sense of LIDO, i. e. they may not be used directly as tree grammar symbols. `INHERITS` constructs like the above are needed to bind their computations to symbols of user's specification. This means avoids accidental coincidence between names of tree grammar symbols and of module roles.

Further `.lido` specifications may be neccessary to supply information to or obtain information from the thus inherited computations.

Modules may also provide C functions (directly or via specifications for other tools), e.g. functions of the environment module in case of the `AlgScope` module. They may be called in computations of the user's `.lido` specification, or in user's C modules if the appropriate header file is included.

The same instantiation mechanism may be applied to include user defined modules:

```
/user/Lib/ModName.gnrc:inst
```

instantiates the module `ModName` of a user library.

Such a module must contain of at least two files , e.g. `/user/Lib/ModName.fw` containing the module's specifications and `/user/Lib/ModName.gnrc`. The latter is an executable shell script that performs the generic module instantiation. It should have the following form:

```
#!/bin/sh
moddir=`expr $0 : '\(.*\)/.*' \| '.'`

$1 -e "s/|NAME|/$2/g
s/|KEY|/$3/g" "$moddir"/ModName.fw > "$2"ModName.fw
```

The last two lines use `sed` to substitute the `instance` parameter for any occurrence of `|NAME|` in the file `ModName.fw` and the `referto` parameter for any occurrence of `|KEY|`.