Programmers manual MCRL toolset

Bert Lisser (2002, CWI)

Draft October 10, 2002

1 Library `mcrl`

The toolset `mcrl` version `2.12.2` is described here.

1.1 Introduction

The `mcrl` library is a collection procedures written in `C`. Its main purpose is to provide an implementation independent access to a linearised mCRL specification, called LPO. This mCRL specification has the shape of an `ATerm`. The tasks of the library `mcrl` are: *interfacing a rewriter chosen by the end user*, *providing I/O*, *adding extra properties to symbols*, *classifying symbols*, *interfacing standard functions (such as* `T`*,* `not`*,* `and`*)*, *interfacing a mCRL specification*, *classifying functions and sorts*, and *adding extra sorts, functions and rewrite rules*. The names of the accessable functions of the `mcrl` library have all prefix `MCRL` or `RW`. Moreover there is access to the structured variables `MCRLsym` and `MCRLterm` containing respectively the symbols of the standard functions with arity greater than 0, and the terms of the standard functions of arity 0.

1.2 Interfacing a rewriter chosen by the end user

For not advanced usage of the `mcrl` library it is satisfiable to initialize the library by one of the following functions: `MCRLinitOnly`, `MCRLinitSU`, or `MCRLinitRW`. In most applications will be used the `mcrl` library in combination with a rewriter. Substitutions will be done during rewriting. There exists a special rewriter which doesn't make use of any rewrite rules. This rewriter performs only substitutions. This section deals about use of the `mcrl` library in combination with a rewriter. It is not possible to use more than one rewriter at the same time.

1.2.1 Initialising `mcrl` library with rewriter

The `mcrl` library will be initialized in combination with a special rewriter which performs only substitutions by

`ATbool MCRLinitSU(int argc, char *argv).`

The `mcrl` library will be initialised in combination with a rewriter determined by the argument after the `-alt` flag if present in `*argv` by

`ATbool MCRLinitRW(int argc, char *argv).`
By default the jitty interpreting rewriter of Jaco van de Pol is used. Both functions returns `T` if all arguments of `*argv` are used and the initialisation is successfull, `F` otherwise. The source file must contain the line

```
#include "rw.h"
```

on top and the loader must be invoked with the flag `-lmcrl`.

1.2.2 Substitution during rewriting

In addition to allocation of entries in the symbol table (see ??), also entries in the substitution table must be allocated for variables. The substitution table is a table of terms indexed by variable symbols. If a term is assigned to a variable, then that value will be stored in the entry of the substitution table belonging to that variable symbol. Initially to each variable is assigned the term containing its variable name. This term is called the *default value* of that variable. Therefore *each variable which occurs in a term must be declared first* by the procedure
`RWdeclareVariables(ATermList varlist).`
The list *varlist* must have the format [$\mathtt{v}(name_1, sort_1)....\mathtt{v}(name_n, sort_n)$]. A term to a variable will be assigned by
`void RWassignVariable(AFun varname, ATerm term, ATerm sort, int level).`
Moreover that term will added to a list of variables which is identified with *level*. It is permitted to assign `NULL` to parameter *sort*. The procedure
`void RWresetVariables(int level)`
sets the variables in the list *level* to its default value and clears this list. Remark: The value of *level* must be greater than 0.
The function
`ATerm RWgetAssignedVariable(AFun var)`
returns the term which is assigned to variable *var*.
The function
`ATerm RWrewrite(ATerm t)`
returns the result of the substitution on term *t*. If the library `mcrl` is initialised with `MCRLinitSU` then the result is not rewritten. However, if the library `mcrl` is initialised with `MCRLinitRW` then the rewritten result will be returned. The function
`ATermList RWrewriteList(ATermList l)`
returns a list from rewritten elements of *l*. Rewriting will be done by calling `RWrewrite` internally. The procedure
`RWflush(void)`
empties the hash table, which is used during rewriting. `RWflush` must be invoked after reassigning a new value to a variable and before calling `RWrewrite` or `RWrewriteList` again.

1.2.3 Initialising `mcrl` library without rewriter

The `mcrl` library without rewriter will be initialized by
`ATbool MCRLinitOnly(int argc, char *argv).`
It is not possible to call functions whose name starts with the prefix `RW`. So rewriting and performing substitutions is not possible. This function returns `T` if all arguments of `*argv` are used and the initialization is successfull, `F` otherwise. The source file must contain the line

```
#include "mcrl.h"
```

on top and the loader must be invoked with the flag `-lmcrl`.

1.3 Providing I/O

1.3.1 Reading arguments from the command line

First the arguments intended for a rewriter will be read and removed from `argv`. Then the arguments intended for the initialisation of `libmcrl` will be read and removed from `argv`. The last element of the remained `argv` is the name of the input file, eventually extended with `.tbf`, which contains the specification. If there is only one argument `argv[0]` or the last element of `argv` starts with a hyphen and there is no previous call of `MCRLsetOutputfile` then there will be assumed that the specification can be read from the stream `stdin`. The procedure
`MCRLsetOutputfile(char output_file[])`
forces that the name of the input file must be supplied by the end user. If the end user forgets this then an error will be generated. The array `output_file[]` contains the name of the input file supplied by the user without suffix `.tbf`. However if the end user invokes a tool with the option `-o` *file_name*, then `output_file[]` contains *file_name* without suffix `.tbf`. `MCRLsetOutputfile` must be called before the initialisation of the `mcrl` library. The function
`char *MCRLgetOutputfile(void)`
returns the address of array *output_file*, which is assigned to the library by `MCRLsetOutputfile`. If `MCRLsetOutputfile` is not called, then NULL is returned.
The following options can be entered on the command line:

-ascii. Writes `.tbf` file in ascii code.

-taf. Writes `.tbf` file in shared representation.

-baf. Writes `.tbf` file in binary format.

-mcrl-hash. Uses hash table during printing.

-extend-adt. The abstract data type will be extended with standard functions and rewrite rules. The end user may have forgotton to supply them to the abstract data type.

-may-extend-adt The abstract data type will not be extended, however if it is unevitable then an minimal extension will be done (default). The tools `structelm` and `sumelm` may extend the abstract data type.

-no-extend-adt. The abstract data type will not be extended; neither with functions, nor with rewrite rules.

-verbose. Prints information messages during run.

-add *file_name***.** Reads extra rewrite rules (in `.mcrl` format) from *file_name* and adds them to the specification.

-rem *file_name***.** Reads extra rewrite rules (in `.mcrl` format) from *file_name* and deletes them from the specification.

Procedure
`MCRLhelp(void)`
prints help information about these options on `<stderr>` .

1.3.2 Reading `.tbf` files

One of the functions `MCRLinitOnly, MCRLinitSU, MCRLinitRW` reads the specification in the file whose name (eventually extended with `.tbf`) is given by the end user in the last argument of the command line. After initialisation the library `mcrl` has access to the whole specification (data part and process part). That specification is called here *current specification.*

1.3.3 Writing `.tbf` files

The current specification can be written by the procedure
`MCRLoutput(void).`
It writes the current specification on `stdout`. The default format in which it is written is the binary format. It is also possible to write this in *shared* format, the end user has to supply the flag `-taf`, or to write this in *ascii* format (flag `-ascii`).

1.3.4 Reading an `MCRLterm` from a string buffer

An `MCRLterm` is a data/action term represented in the format which belongs to the toolset. The function
`ATerm MCRLparse(char *e)`
reads a term in natural form from e, and returns the belonging MCRLterm (for example `char*: S(0) -> ATerm: S#Nat(0#)`).

1.4 Adding extra properties to symbols

To each *function*, *sort*, and *variable* belongs an unique *symbol.* Each symbol can be identified by its *long name* and its *arity.* The symbols belonging to sorts and variables have arity 0. The arity of a symbol belonging to a function is equal to the arity of the *function*. A symbol table which is indexed by that symbol is maintained by library `mcrl`. This symbol table contains information about sorts, functions, and variables. In addition to the already earlier mentioned *name* and *arity* the following information about a symbol is available: its *short name*, its *type* and, if the symbol itself is not a represention of a sort, its *signature* and its *sort*.

1.4.1 Short name of a symbol

In a `.tbf` file has each function or sort an unique long name. That name is includes its *short name* defined by the end user and if applicable the names of the sorts of its arguments. The function

`char *MCRLgetName(ATerm t)`

returns the short name of the head symbol of term t. The function

`char *MCRLextendName(char *short_name, ATermList sorts)`

returns the long name of the symbol determined by the short name and the sorts of its arguments. The functions

`ATerm MCRLprint(ATerm term)`

and

`ATermList MCRLprintList(ATermList list)`

convert respectively *term* and *list* to an ATerm and an ATermList which are suitable to be printed. The long names of the subterms are changed into their short names.

1.4.2 Type

To each symbol is assigned a type. The following types are available: `MCRLunknown`, `MCRLsort`, `MCRLconstructorsort`, `MCRLenumeratedsort`, `MCRLvariable`, `MCRLconstructor`, `MCRLfunction`, `MCRLcasefunction`, and `MCRLeqfunction`. `MCRLsymtype` is the enumeration of these types. The function

`MCRLsymtype MCRLgetType(AFun sym)`

returns the type of *sym*. If there is not assigned a type to a symbol then `MCRLgetType` returns `MCRLunknown`. The procedure

`MCRLsetType(AFun sym, MCRLsymtype type)`

assigns *type* to *sym*.

1.4.3 Signature

The term $f(sort_1, \ldots, sort_n)$, which is an application of symbol f on the sorts of its arguments, is assigned to each symbol f belonging to a function. The term which exists only of the symbol f is assigned is assigned to each symbol belonging to a constant (function with arity 0) or variable. The function

`ATerm MCRLgetFunction(AFun f)`

returns this term. If f belongs to a sort this function returns *null*. The procedure

`MCRLsetFunction(ATerm sig, MCRLsymtype type, AFun sort)`
assigns the signature *sig*, *type*, and *sort* to the header symbol of *sig*. The function

`ATerm MCRLuniqueTerm(char *short_name, ATermList sorts)`
returns the signature with *long_name* as header symbol if *long_name* is not already present in the current specification. If *long_name* already present then an unique signature with its *short_name* is suffixed by $'\langle n\rangle$ will be returned.

1.4.4 Sort

A sort is assigned to each symbol f which is not a symbol of a sort. The functions
`AFun MCRLgetSort(ATerm t)`
and
`AFun MCRLgetSortSym(AFun f)`
return respectively the sort of a term and the sort of a function symbol. To each sort belongs a dummy value of that sort, which serves as a kind of *don't care* value. This value will be used in the tools `mcrl` and `structelm`. The function
`MCRLdummyVal(AFun sort)`
returns that value.

1.5 Classifying symbols

The symbols will be classified in *function symbols*, *sort symbols*, and *variable symbols*.

1.5.1 Function symbols

In this section the symbols belonging to functions will be described. Already mentioned is that to a symbol f is connected its *long name*, its *arity* n, its *sort*, its *class*, and its *signature*. In addition there is added a *list of rewrite rules* of which f is the header symbol of the left hand side. The function
`ATermList MCRLgetRewriteRules(AFun fsym)`
returns this list.
A long name of a function has the following format

$$short\_name\#sort_1 \ldots \#sort_n.$$

Each function name must contains an #.

Each function symbol is classified in one of the following classes: `MCRLconstructor, MCRLeqfunction, MCRLcasefunction`, or `MCRLfunction`. The language `mcrl` classifies functions in `MCRLconstructor` and `MCRLfunction`. The functions of the first class appear directly after the key word `fun`, the others appear directly after the key word `map`.

1.5.2 Sort symbols

In this section the symbols belonging to sorts will be described. Already mentioned is that to a sort symbol s is assigned its *long name*, its *arity* n, its *type*. In addition there is assigned a *list of its constructors*. Function

`ATermList MCRLgetConstructors(AFun s)`

returns this list. Format of this list:

$$[cons_1(sort_1^1, \ldots, sort_1^{n_1}), \ldots, cons_m(sort_m^1, \ldots, sort_m^{n_m})].$$

A long name of a sort is a string which does not contain #. A sort and a function in a specification can have the same short name, but their long names are different. For example: a function without arguments with *short name* `tuple` has *long name* `tuple#` and a sort with *short name* `tuple` has *long name* `tuple`.

Each sort is classified into one of the following classes: `MCRLconstructorsort`, `MCRLenumeratedsort`, or `MCRLsort`.

1.5.3 Variable symbols

To a variable symbol v is assigned a *name*, the arity 0, the type `MCRLvariable`. A name of the variable has the following format *short_name#*. It is not possible that a variable and a function with the same name both exist. The procedure

`MCRLdeclareVars(ATermList varlist)`

allocates entries in the symbol table belonging to the the variables in *varlist*. Each entry obtains the name of the variable, arity 0, and the sort of the variable. Format of *varlist*: [`v`($name_1$, $sort_1$)....`v`($name_n$, $sort_n$)]. If the library `mcrl` is initialised with `MCRLinitOnly` then all variables must be declared first by `MCRLdeclareVars`, before using them. If the library *mcrl* is initialized with `MCRLinitSU` or `MCRLinitRW` then `RWdeclareVariables` must be used instead of `MCRLdeclareVars` (`RWdeclareVariables` calls `MCRLdeclareVars`).

There is available a function

`ATermList MCRLremainingVars(ATerm t, ATermList varlist)`

which deletes from a list of variables *varlist* the variables which occurs in a given term t. If *varlist* is the list of variables which are expected in term t, then it is possible to determine if the term t is a closed term or not by the test

```
ATisEqual(MCRLremainingVars(t, varlist), varlist).
```

Format of *varlist* as in `MCRLdeclareVariables`.

1.6 Interfacing standard functions

There will be assumed that in a mCRL specification are defined the functions `T`, `F`, `not`, `and`, `or`, `eq`, and `if`. If the flag `extend-adt` is supplied by the end user, then the lacking functions will be generated with their belonging rewrite rules, otherwise a warning will be printed. The head symbols belonging to the standard functions with arity greater than 0 can be accessed via the variable

MCRLsym, the head symbols belonging to the standard functions with arity 0 can be accessed via the variable MCRLterm. The following functions and equations will be checked on presence; if the flag -extend-adt is suppied then the missing functions and rewrite rules will be added to the specification.

1. Constructors $\mathtt{T}, \mathtt{F} \mapsto \mathtt{Bool}$. If not present in mCRL specification then even with the -extend-adt flag an error will be generated. The terms MCRLterm.true and MCRLterm.false contain respectively "T#" and "F#".

2. Map not : $\mapsto$ Bool and the following rewrite rules.

$$\begin{array}{ll} \mathtt{var} & x : \mathtt{Bool} \\ \mathtt{rew} & \mathtt{not}(\mathtt{T}) \quad = \quad \mathtt{F} \\ & \mathtt{not}(\mathtt{F}) \quad = \quad \mathtt{T} \\ & \mathtt{not}(\mathtt{not}(x)) \quad = \quad x \end{array}$$

   The symbol MCRLterm.not gets name "not#Bool" and arity 1.

3. Map and : $\mathtt{Bool} \times \mathtt{Bool} \mapsto \mathtt{Bool}$ and the following rewrite rules.

$$\begin{array}{llcl} \mathtt{var} & x : \mathtt{Bool} & & \\ \mathtt{rew} & \mathtt{and}(x, \mathtt{F}) & = & \mathtt{F} \\ & \mathtt{and}(\mathtt{F}, x) & = & \mathtt{F} \\ & \mathtt{and}(\mathtt{T}, x) & = & x \\ & \mathtt{and}(x, \mathtt{T}) & = & x \end{array}$$

   The symbol MCRLsym.and gets name "and#Bool#Bool" and arity 2.

4. Map or : $\mathtt{Bool} \times \mathtt{Bool} \mapsto \mathtt{Bool}$ and the following rewrite rules.

$$\begin{array}{llcl} \mathtt{var} & x : \mathtt{Bool} & & \\ \mathtt{rew} & \mathtt{or}(x, \mathtt{F}) & = & x \\ & \mathtt{or}(\mathtt{F}, x) & = & x \\ & \mathtt{or}(\mathtt{T}, x) & = & \mathtt{T} \\ & \mathtt{or}(x, \mathtt{T}) & = & \mathtt{T} \end{array}$$

   The symbol MCRLsym.or gets name "or#Bool#Bool" and arity 2.

5. If-then-else map : There will be looked for a map *if* : $\mathtt{Bool} \times \mathtt{Bool} \times \mathtt{Bool} \mapsto \mathtt{Bool}$ with the following rewrite rules in the mCRL specification:

$$\begin{array}{llcl} \mathtt{var} & x, y : \mathtt{Bool} & & \\ \mathtt{rew} & \mathit{if}(\mathtt{T}, x, y) & = & x \\ & \mathit{if}(\mathtt{F}, x, y) & = & y \end{array}$$

   The symbol MCRLsym.ite gets the name of that map *if*.

If not such a map is present and the -extend-adt flag is given, then such a map and its belonging rules will be added to the specification under an unique name. Moreover the extra rule $\mathit{if}(x, y, y) \;=\; y$ will be added.

1.7 Classifying functions and sorts

In this section will be described in chronological order the process of assigning a type to the functions and sorts defined in an mCRL specification. This will happen during initialisation. Assumed is that to all functions is already assigned the type `MCRLconstructor` or `MCRLfunction`.

1.7.1 Assigning the type `MCRLsort` to sort s

If type `MCRLsort` is assigned to sort s then s not a result sort of any constructor. To the remaining sorts will be assigned temporary the type `MCRLconstructorsort`.

1.7.2 Assigning the type `MCRLeqfunction` to function f

To function symbol f is assigned the type `MCRLeqfunction` iff the following hold:

1. Function symbol f has the type `MCRLfunction`.
2. The short name of f is equal to `eq`.
3. Arity f is equal to 2 .
4. The sorts of its two arguments must be equal.
5. The result sort of f is `Bool`.
6. There exists a rewrite rule with left hand side is equal to `eq(x,x)`, where x is a variable, and the rhs is equal to `T`.

1.7.3 Assigning the type `MCRLenumeratedsort` to sort s

To sort symbol s is assigned the type `MCRLenumeratedsort` iff the following holds.

1. Sort symbol s has type `MCRLconstructorsort`.
2. All constructors of s are constant functions (arity 0).
3. There is no rewrite rule defined where the header symbol of its left hand side is equal to a constructor of sort s.
4. There exists an `eq` function for which holds that each pair of different constructors (c_i, c_j) is presented by the rewrite rule $\mathtt{eq}(c_i, c_j) = \mathtt{F}$ in the specification or the `eq` function doesn't exist at all for sort S. In the last case and the tool is called with the flag `-extend-adt` then the map $\mathtt{eq} : \mathtt{S} \times \mathtt{S} \mapsto \mathtt{Bool}$ and the following rewrite rules:

$$\begin{array}{lllll} \mathtt{var} & x : \mathtt{Bool} & & \\ \mathtt{rew} & \mathtt{eq}(x, x) & = & \mathtt{T} \\ & \mathtt{eq}(c_i, c_j) & = & \mathtt{F},\ c_i \neq c_j,\ 1 \leq i, j \leq n \end{array}$$

   where $c_i, 1 \leq i \leq n$ are the (constant) constructors of `S`, will be added.

1.7.4 Assigning the type `MCRLcasefunction` to function f

To function symbol f is assigned the type `MCRLcasefunction` iff the following holds.

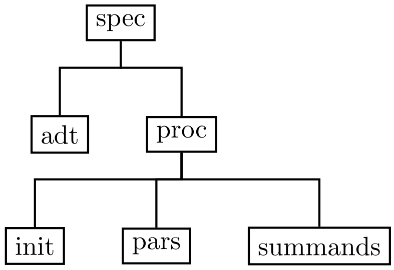
1. Function symbol f has the type `MCRLfunction`.
2. The first argument of function f has an enumerated sort E of cardinality n.
3. Function f has $n+1$ arguments.
4. The sorts of argument 2 until argument $n+1$ and the result sort S are all equal.
5. For each constructor c_i of the enumerated sort of argument 1 there exists one rewrite rule $f(c_i, x_1, \ldots, x_n) = x_{k_i}$, where $x_1, \ldots, x_n$ are all variables of sort S. All $x_{k_i}, 1 \leq i \leq n$ must be different from eachother. The list $[c_{k^{-1}(1)}, \ldots, c_{k^{-1}(n)}]$ is called here the selectors of f.
6. There are two cases
   (a) The sequence of the constructors of the enumerated type E is not fixed. The constructors will be reordered. Let $k : 1 \ldots n \mapsto 1 \ldots n$ be a bijection. Then the sequence of the constructors $[c_1, \ldots, c_n]$ will be changed to the sequence of the selectors $[c_{k^{-1}(1)}, \ldots, c_{k^{-1}(n)}]$. The sequence of the constructors of E will be marked as fixed.
   (b) The sequence of the constructors of the enumerated type E is already fixed. The sequence of selectors must be equal to the sequence of constructors.

The selectors of a case function f can be obtained by
`ATermList MCRLgetCaseSelectors(AFun f).`
A casefunction will be added to the list of case functions which is assigned to its result sort symbol. The function
`MCRLgetCasefunctions(AFun sortsym)`
returns this list. For each constructor sort there exists a special casefunction, which will be placed at the head of the list. The cardinality n of the enumerated sort of the first argument of this casefunction is equal to the number of constructors of S. The tool `structelm` uses these special casefunctions. Casefunctions have nice properties, which optimise rewriting.

1.8 Interfacing mCRL specifications

1.8.1 Structure of mcrl specification

In the `.tbf` file is written the whole specification as an `aterm`. The whole specification is divided into two parts: the *abstract data type* (`adt`) and the *process definition* (`proc`). The process definition is divided into three parts: the *intialisation* *vector* (`init`), the *process parameters* (`pars`), and the *summands* (`summands`).

This structure is expressed as the following tree: 

1.8.2 Updating parts of the specification

Here follow the procedures which are designed for updating the specification. The function

`ATerm MCRLgetSpec(void)`

returns the current mCRL specification. The procedure

`MCRLsetSpec(ATerm spec)`

replaces the current specification with mCRL specification *spec*. The function

`ATerm MCRLgetAdt(void)`

returns the abstract data type belonging to the current specification. The procedure

`MCRLsetAdt(ATerm adt)`

replaces the abstract data type belonging to the current specification with *adt*. The function

`ATerm MCRLgetProc(void)`

returns the process belonging to the current specification. The procedure

`MCRLsetProc(ATerm proc)`

replaces the process belonging to the current specification with *proc*. The function

`ATermList MCRLgetListOfInitValues(void)`

returns the list of initial values belonging to the process of the current specification. The function

`ATermList MCRLgetListOfPars(void)`

returns the list of data parameters in the process definition. The function

`ATermList MCRLgetNumberOfPars(void)`

returns the number of data parameters in the process definition. The function

`ATermList MCRLgetListOfSummands(void)`

returns the list of summands of belonging to the process of the current specification. The function

`int MCRLgetNumberOfSummands(void)`

returns the number of summands belonging to the process of the current specification. The procedure

`MCRLsetListOfSummands(ATermList summands)`

replaces the list of summands belonging to the process of the current specification with *summands*. The function

```
ATermList MCRLgetListOfVars(ATerm summand)
```
returns the list of data variables belonging to *summand*. The function
```
int MCRLgetNumberOfVars(ATerm summand)
```
returns the number of data variables belonging to *summand*.

1.8.3 Adding objects to the abstract data type

The API programmer has at his disposal routines to extend the *current abstract data type* with sorts, functions, and equations. These routines will, check first if the object that has to be added is already present. The result of that check will be stored in parameter `*new`. In case that `new` is the `null` pointer only the check on presence will be done. If the object is not present then it is considered as an error, which means that nothing will be added and 0 will be returned. The function
```
AFun MCRLputSort(ATerm sort, ATbool *new)
```
adds a new sort to the current specification and assigns `ATtrue` to `*new` if *sort* is not present in the specification, and assigns `ATfalse` to `*new` otherwise. Normally the header symbol belonging to *sort* will be returned, 0 will be returned in case of error. However if `new` is equal to `null` and *sort* is not already present it is an error. The function
```
AFun MCRLputConstructor(ATerm sig, ATerm rsort, ATbool *new)
```
adds a new constructor with result sort *rsort* to the specification and assigns `ATtrue` to `*new` if *sig* is not present as function in the specification, and assigns `ATfalse` to `*new` otherwise. Normally the header symbol belonging to *sig* will be returned, 0 will be returned in case of error. However if `new` is equal to `null` and *sig* is not already present then it is an error. The parameter `sig` must have the format $f(sort_1, \ldots, sort_n)$ and *rsort* must contain an existing sort. The function
```
AFun MCRLputMap(ATerm sig, ATerm rsort, ATbool *new)
```
adds a new map with result sort *rsort* to the specification and assigns `ATtrue` to `*new` if *sig* is not present as function in the specification, and assigns `ATfalse` to `*new` otherwise. Normally the symbol belonging to *sig* will be returned, 0 will be returned in case of error. However if `new` is equal to `null` and *sig* is not already present then it is an error. The parameter `sig` must have the format $f(sort_1, \ldots, sort_n)$ and *rsort* must contain an existing sort. The function
```
AFun MCRLputEquation(ATerm eq, ATbool *new)
```
adds a new equation to the specification and assigns `ATtrue` to `*new` if *eq* is not present as function, and assigns `ATfalse` to `*new` otherwise. Normally the header symbol belonging to *eq* will be returned, 0 will be returned in case of error. However if `new` is equal to `null` and *eq* is not already present it is an error. The parameter `eq` must have the format $\mathtt{e}([\mathtt{v}(name_1, sort_1) \ldots, \mathtt{v}(name_n, sort_n)], lhs, rhs)$, where *rhs* and *lhs* are terms.

1.8.4 Adding standard functions to the abstract data type

The API programmer has at his disposal routines to add standard functions with their belonging equations to the abstract data type. These routines return normally its symbol and return 0 on error. If the function is already present in the specification then ATfalse is assigned to the boolean parameter *new. In that case these routines don't add missing equations to the specification. If the function is not present in the specification then ATtrue is assigned to the boolean parameter *new and this function is added and its standard rewrite rules are added to the specification. However if the flag -no-extend-adt is used or parameter new is equal to null, then it is forbidden to add new functions and equations. If the wanted function is not found, then these functions return 0. There is a distinction between calling these routines with ATtrue assigned to *new and ATfalse assigned to *new. If a routine is called with ATfalse assigned to *new then the abstract data type will be only extended with equations if the concerning function is absent and therefore it will be added, however if a routine is called with ATtrue assigned to *new then the abstract data type will be extended with lacking equations, even if the concerning function is already present. Here follow the headers of these routines.

AFun MCRLputNotFunction(ATbool *new)
adds not and its rewrite rules to the *current specification*.

AFun MCRLputAndFunction(ATbool *new)
adds and and its rewrite rules to the *current specification*.

AFun MCRLputOrFunction(ATbool *new)
adds or and its rewrite rules to the *current specification*.

AFun MCRLputEqFunction(ATerm sort, ATbool *new)
adds eq#<sort>#<sort> and its rewrite rules to the *current specification*. If *sort* is not an enumerated sort, then MCRLputEqFunction generates an error and returns 0.

AFun MCRLputCaseFunction(int n, ATerm rsort s, ATbool *new)
adds a new case function with result sort rsort and its rewrite rules to the *current specification*. First there will be looked for a case function of arity $n+1$ belonging to sort rsort. If such a case function is not found, then a new one will be created. For that purpose is needed an enumerated sort s of cardinality n. If sort s is not defined then an enumerated sort s of cardinality n will be created. The sequence of selectors is equal to the sequence of constructors of s.

AFun MCRLputIfFunction(ATerm rsort, ATbool *new)
adds a new if-then-else function if with result sort rsort, its required equations if(T,x,y)=x and if(F,x,y)=y, and its extra equation if(b, x, x)=x to the *current specification*, in which x and y are variables of sort rsort and b is a variable of sort Bool. First there will be searched a case function *if* of arity 3 belonging to sort *rsort* which must have the required equations $\mathit{if}(\mathtt{T}, x, y) = x$ and $\mathit{if}(\mathtt{F}, x, y) = y$, in which x and y are variables of sort rsort. If such a function is not found then there will be searched a function if#Bool#<rsort>#<rsort> which must have the required rewrite rules. If such a function is found then its function symbol will be returned.

Moreover if `ATtrue` is assigned to `*new` then the lacking rewrite rules will be added. In this case the remaining rule `if(e,x,x)=x`.
If such a function is not found then a `if-then-else` function with its rewrite rules will be created and added to the current specification. It is not self-evident that this is a `casefunction`. The sequence of selectors [`T,F`] does not need to be equal to the sequences of selectors belonging to casefunctions of arity 3.

2 Adding a new rewriter to the `mcrl` interface

2.1 Requirements

The added rewriter have to be able to rewrite open terms and have to be able to perform substitutions. The arguments and result values have to be `aterms`. The signature and rewrite rules are stored in an `aterm`. So the `aterm` library is needed.

2.2 Outline about what must be done

Suppose a rewriter called `XX` will be added. The following things must be done.

1. Create a file libxx.c which contains the line `#include "tasks.h"`.
2. Define a new structured variable `XXtasks` which contains the implementation of the rewriter `XX`. This variable will be exported.
3. Update the procedure `RWsetArguments` in `tasks.h`.
4. Update the procedure `RWhelp` in `tasks.h`.
5. Add `libxx.o` to particular lines in the section `libmcrl.a:` of `Makefile.in`.

3 How the libraries used in the toolset are related to eachother

3.1 Dependency tree

The toolset uses the libraries `libaterm.a (aterm2.h), libmcrl.a (mcrl.h, rw.h), libstep.a (step.h), liblts.a (lts.h) , and libsvc.a (svc.h)` . To each library name is appended between brackets the name of its belonging include file. All libraries use the aterm library. If a library can be reached from an other library, then this library uses this library. So all libraries use the mcrl library, because all libraries can be reached from the mcrl library. Here is depicted a dependency tree of the libraries which the toolset uses. The system libraries and the aterm library are omitted. The rectangles are libraries, the ellipses are tools.

The library mcrl includes a set of rewriters and provides for an interface to one of these, by the user chosen, rewriters. The box $mcrl + rww$ refers to access

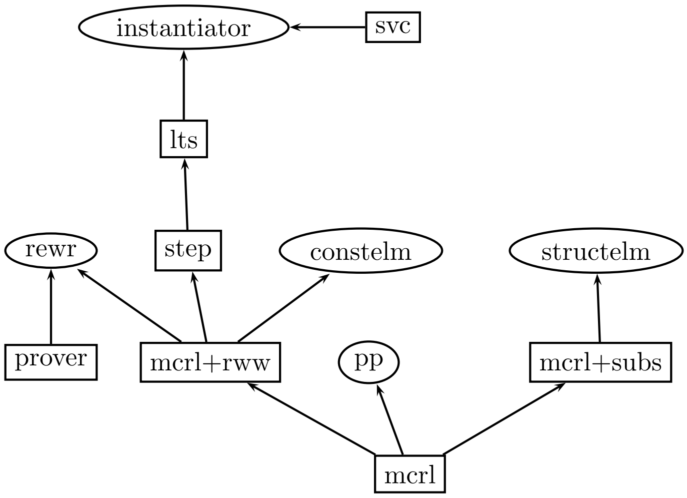


Figure 1: Hierarchy of libraries

to the mcrl library and one rewriter chosen by the enduser, the box *mcrl + subs* refers to access to the mcrl library and the substitution mechanism.

3.2 Initialisation of these libraries

Each library must be initialised. The initialisation consists of three phases

initialisation aterm library: This is a call of `ATinit(argc, *argv, &argc);`.

eating arguments: A leave of the dependency tree, which is a tool, must eat his arguments first, then his parent must eat his arguments, until there are only arguments left for the mcrl library and the aterm library. For example

```
int main(int argc, char *argv[]) {
    int i, j = 0;
    char **newargv = (char**) calloc(argc, sizeof(char*));
    ATinit(argc, argv, (ATerm*) &argc);
    newargv[j++] = argv[0];
    for (i=1;i<argc;i++) {
         if (!strcmp(argv[i],<option>)) {
            /* Do something */
            continue;
            }
         .
         .
         newargv[j++] = argv[i];
         }
```

```
LTSsetArguments(&j, &newargv);
```

The path along which the `...setArguments` is invoked goes from leave to root. The reason for this order of invocation, is that the remaining last argument if it exists will be offered to the mcrl library as input file name. LTSsetArguments eats also arguments belonging to the step library. These arguments will be passed to the step library.

allocating and initialising memory: The root which is the mcrl library must be initialised first, then the child, and so on, until the wished leave is reached.

```
if (!MCRLinitRW(j, newargv)) exit(1);
LTSinitialize(...);
Instantiator();
}
```

The library libmcrl.a contains all rewriter objects. Each rewriter object must be chosen and initialized before using it. In fact `MCRLinitRW` is defined as

```
RWsetArguments(&j, &newargv);
MCRLsetArguments(&j, &newargv);
if (/* The remaining arguments are no arguments of the aterm library */)
        error(..)
MCRLinitialize();
RWinitialize();
```

The path along which the `...initialize` is invoked goes from root to leave.

4 Library *step*

5 Library *lts*

6 Tool *instantiator*, generation of state space

7 About compiling and linking

Reference to functions