Centrum voor Wiskunde en Informatica

REPORTRAPPORT

SEN

Software Engineering

Software ENgineering

SEN

Manual for the μCRL tool set (version 2.8.2)

A.G. Wouters

REPORT SEN-R0130 DECEMBER 31, 2001

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Manual for the μCRL Tool Set (version 2.8.2)

Arno Wouters
Email: `Arno.Wouters@cwi.nl`

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
The specification language μCRL and a tool set for manipulation, optimisation and state space generation are described.

2000 Mathematics Subject Classification: 68-00,68N30,68Q60,68Q85
2000 ACM Computing Classification System: A.2,D.2.2,D.2.4,F.3.1
Keywords and Phrases: formal methods, specification, verification, process algebra, tools

1. Introduction

The μCRL tool set [BFG+01] is a collection of tools for analysing systems of communicating processes described in μCRL (micro Common Representation Language). An overview of the tool set is given in Fig. 1.

`mcrl`	• checks whether a specification in (timed) μCRL is well formed, • linearises certain μCRL specifications.
`msim`	allows interactive simulation of a system described in μCRL.
`instantiator`	generates a finite transition system from a linearised μCRL specification.
`pp`	pretty prints a linearised μCRL specification.
`rewr`	normalises the data terms in a linearised μCRL specification.
`constelm`	removes from a linearised μCRL specification the data parameters that are constant throughout any run of the process.
`parelm`	removes from a linearised μCRL specification the data parameters and sum variables that do not influence the behaviour of the system.
`structelm`	expands the composite data types of a linearised μCRL specification.
`sumelm`	replaces in a linearised μCRL specification the sum variables that must be equal to a certain data term by that data term.

Figure 1: Overview of the μCRL tool set

μCRL [GP95] is a language to describe communicating processes. It is based on the process algebra ACP [BW90, Fok00] extended with equationally specified data types [LEW96]. Despite its simplicity it is quite adequate to specify and analyse (large) distributed systems and algorithms. μCRL has been extended with features to express time [Gro97], but the tools do not support this extension, except for the possibility to check the static semantic constraints.

The tool set is constructed around a restricted form of μCRL, namely the linear process operator format (LPO) [BG93].[1] The tool `mcrl` checks whether a certain specification is well formed μCRL and attempts to transform it into a linearised (i.e. LPO) form. This linearised form is stored in binary form (more precisely in binary aterm format, also called tool bus format (`.tbf`)). All other tools use this linearised format as their starting point.

These tools come in four kinds:

1. a tool (`msim`) to step through a process described in μCRL,
2. a tool (`instantiator`) to generate a transition system in a format (`.aut`) that can be read by the model checker CÆSAR ALDÉBARAN.
3. several tools to optimise the linearised specification:
   (a) `rewr`, normalises the data terms in a specification by performing the rewritings it specifies,
   (b) `constelm`, removes data parameters that are constant throughout any run of the process,
   (c) `parelm`, reduces the state space of the transition system by removing the data parameters and sum variables that do not influence the behaviour of the system,
   (d) `structelm`, expands variables of compound data types,
   (e) `sumelm`, replaces sum variables that must be equal to a certain data term by that data term.
4. a tool (`pp`) to print the linearised specification.

An overview of the relations between the tools in the tool set is sketched in Fig. 2.

This is a typical simple session with the tool set:

1. Write the specification with your favourite editor:

   `vi spec`

2. Check well-formedness and linearise the specification with `mcrl`:

   `mcrl -tbfile -regular spec`

   (see Section 3.1 for an explanation of the flags)

3. Generate a state space with `instantiator`:

   `instantiator -i spec`

   (see Section 3.3 for an explanation of the flags)

4. Study the state space with CÆSAR ALDÉBARAN:

   `xeuca &`

This manual aims to provide enough information to use the tools of the tool set. It assumes a basic knowledge of process algebra and abstract data types. You are now reading the introductory section of this manual (Chapter 1). Chapter 2 describes the language μCRL as it is recognised by the tool set in an informal way. Chapter 3 describes the functionalities, options and shortcomings of each tool. Appendix I contains a formal specification of timed μCRL. Appendix II sketches the LPO format and the linearisation procedure. Appendix III discusses the rewriting method. Appendix IV provides the text of the license agreement that covers the use of the tool set.

More information:

[1] see Section II.1.

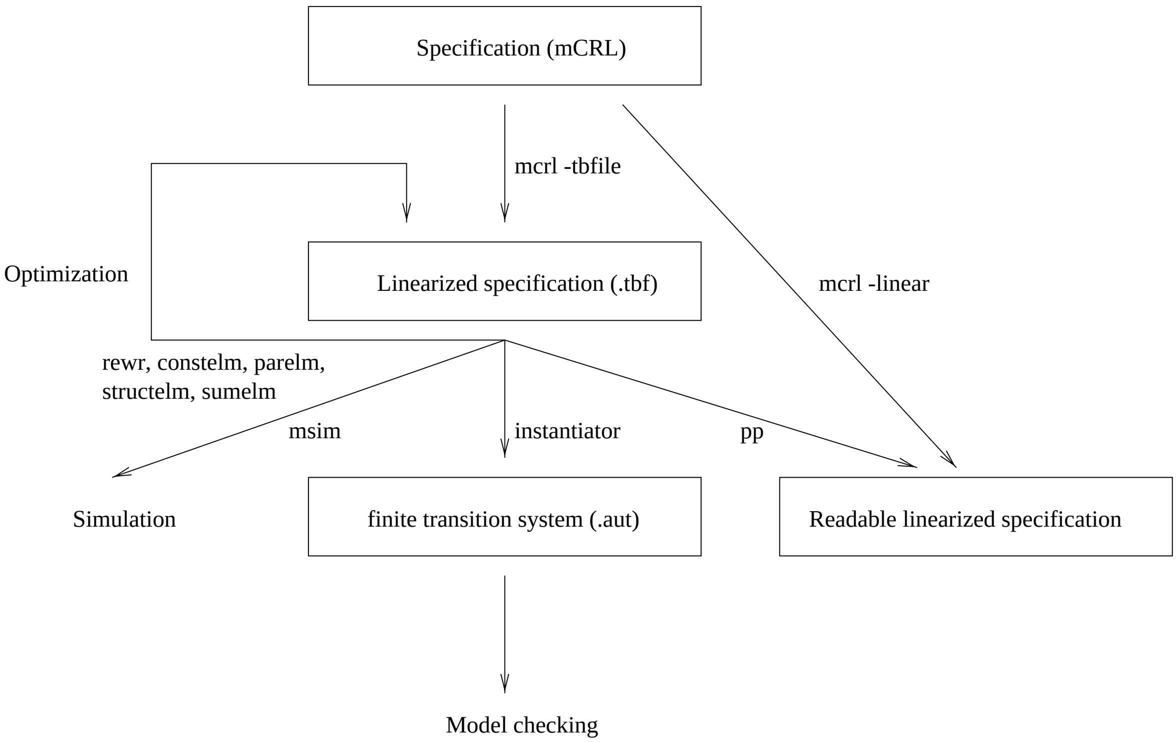

Figure 2: The main relations between the tools in the μCRL tool set

- The lecture notes of Wan Fokkink, Michel Reniers, & Jan Friso Groote (*Modelling Concurrent Systems: Protocol Verification in μCRL*) at `<http://www.cwi.nl/~wan/lecturenotes.ps>` provide a basic introduction to the use of μCRL. These notes include crash courses in process algebra, abstract data types and protocol verification, as well as a handful of examples of protocol specifications.

- The μCRL tool set is developed by the Embedded Systems group of the Dutch Centre for Mathematics and Computer Science (CWI). The μCRL home page at `<http://www.cwi.nl/~mcrl/>` provides information about the language and the tool set. The tool set is freely available from `<http://www.cwi.nl/~mcrl/mutool.html>`.

- The CÆSAR ALDÉBARAN Development Package is developed by the VASY (Validation of Systems) group of INRIA (France). The CADP home page is at `<http://www.inrialpes.fr/vasy/cadp/>`.

About the tool set

The tool set is developed by Jan Friso Groote (`JanFriso.Groote@cwi.nl`) and Bert Lisser (`Bert.Lisser@cwi.nl`). If a problem is detected it is appreciated if a bug report is sent to the authors. The report should include a description of the problem (including the exact error message) and information needed to repeat the error (such as the version number of the tool set, name and version number of the operating system, exact command line used to invoke the tool (including all flags), and a copy of the input file).

The version numbering policy is this: the number behind the point changes if only a bug is fixed, the number before the point changes if the functionality is extended or the structure of the code improved.

Version history

- Version 1.0, (for tool set version 1.10) *Tutorial and Reference Guide for the μCRL tool set*, by Jan Friso Groote and Bert Lisser, distributed with the tool set.
- Version 2.0 (for tool set version 1.11), *Manual for the μCRL tool set*, August 2000, by Arno Wouters.
- Version 3.0 (for tool set version 2.8.2), *Manual for the μCRL tool set*, 14 December 2001, by Arno Wouters. This manual.

Acknowledgements The author thanks Wan Fokkink, Jan Friso Groote, Izak van Langevelde, Bert Lisser, Alban Ponse and Yaroslav Usenko for their comments on earlier drafts of this manual.

2. μCRL

This chapter summarises μCRL as it is recognised by the tool set. The chapter is meant as a reference for people who know the basics of μCRL and want to use the tool set to specify and analyse systems. It is not meant as a tutorial, neither is it meant as a formal specification of μCRL. A formal specification of the syntax of μCRL is given in appendix I of this manual.

2.1 The basics

Basic structure A μCRL specification consists of two parts. The first part specifies the data types (Section 2.2), the second part the processes (section 2.3).

Names In μCRL there are five kinds of entities that have names, namely: sorts, functions, variables, actions, and processes. The following conditions apply to the use of names:

- Names may consist of letters (`a-z`, `A-Z`), digits (`0-9`) and the special characters `^_'-`.
- All names must be declared.[2]
- Each sort, function, action and process must be declared exactly once.
- Functions, actions, processes and variables may have the same name as a sort.
- Different functions, actions and processes can not have the same combination of name and domain sorts.
- Variables can not have the same name as function constants, parameterless actions or parameterless processes.
- Names of variables must be unique within their declaration (this means that it is allowed to use the same variable name for different sorts in different equation sections, process declarations and sum-operators).

Comments Comments start with a % character and end at the end of the line.

2.2 Specification of data

Data are represented as terms of some sort, for example `S(S(0))`, `cos(pi)`, and `concat(L1,L2)` could be terms of sorts natural number, real number, and list, respectively. Fig. 3 gives an overview of a data specification, Fig. 4 an example.

[2] In the format descriptions of this manual capitalised words are used to indicate that the corresponding name is being declared at that point.

sort Sortname
func Functionname : domain -> sortname
map Functionname : domain -> sortname
var Varname, Varname . . . : sortname
rew data-term = data-term

Figure 3: Data specification format

```
sort Bool
func T,F:                    -> Bool
map  not: Bool               -> Bool
     and, or: Bool # Bool -> Bool

var  bool          -> Bool
rew  not(F)        = T
     not(T)        = F
     and(T, bool)  = bool
     and(F, bool)  = F
     and(bool,F)   = F
     and(bool,T)   = bool
     or(T,bool)    = T
     or(bool,T)    = T
     or(F,bool)    = bool
     or(bool,F)    = bool
```

Figure 4: An example data specification

Sorts Sorts are declared with the keyword **sort**, followed by a name or a list of names (space separated). Each declared sort represents a non-empty set of data. The elements of that set are represented as functions.

Functions Functions are declared with the keywords **func** or **map**:

```
func | map  Functionname-list : domain -> sortname
```

For example:

```
func T, F: -> Bool
map  and: Bool # Bool  -> Bool
```

- The keyword **func** is used to declare the functions that construct a certain sort (the functions that define the elements of that sort).
- The keyword **map** is used to declare operations on the elements of an already defined sort.
- The functionname-list is a comma separated list of names
- The domain is a list of declared sort names separated by hashes (#). Domains can be empty.
- It is allowed to overload names of functions, as long as the sort of each term can be determined uniquely. This means that the combination of the function name and the sorts of its domain must be unique. It is, for example, allowed to define two functions `max` in the following way:

```
map max: Nat  # Nat  -> Nat
    max: Real # Real -> Real
```

but not in the following way:

```
map max: Nat  # Nat  -> Nat
    max: Nat  # Nat  -> Real
```

- A function can not have the same combination of name and domain sorts as an action, process or variable.
- If a sort `D` is declared without a constructor function that sort is assumed to be arbitrarily large. In particular `D` can contain elements that cannot be denoted by terms. As it is not possible to generate state spaces of processes with infinite data sorts, specifications with arbitrarily large domains are not practical.
- It is not allowed to define empty sorts. For example, the following definition of `D` will be rejected:

  ```
  sort D
  func f:D->D
  ```

  According to this definition every element of `D` can be written as an application of `f` onto an element in D. This means that the only element in `D` can be written as an infinite sequence of applications of `f`. As terms are finite objects `D` must be empty, which is forbidden.
- There are no pre-defined data types in μCRL. In other words, all data types must be specified.
- The sort `Bool` must be declared and it must have two constructors, namely `T` and `F`.

Data-terms Data-terms can have the following formats:

```
functionname
functionname ( data-term-list )
varname
```

Data-term-lists are comma separated lists of data-terms. The following conditions apply:

- The names of functions and variables must be declared.
- Data-terms must be well typed with respect to their declaration.

For example, given the following declarations:

```
sort Real Point
func 3:                 -> Real
     point: Real#Real   -> Point
map  cos: Real          -> Real
```

some data-terms are: `3`, `cos(3)`, `cos(cos(3))`, `point(3,3)`.

Equations The additional properties and relations declared with the keyword **map** are defined by means of equations. An equation specification consists of an optional variable-declaration (starting with the keyword **var**) followed by an equation-section (starting with the keyword **rew**).

Variable-declaration Variable-declarations consist of the keyword **var** followed by a (space separated) list of typed variables:

```
var Varname-list : sortname
    Varname-list : sortname
```

The varname-list is comma separated.

- The sorts must be declared.
- Variables can not have the same name as function constants, parameterless actions or parameterless processes.
- The names of variables must be unique within the declaration. This means that it is allowed to use the same name in different equation specifications for different sorts, as in the following example:

  ```
  var  x: Bool
  rew  and(T,x) = x
       and(F,x) = F

  var  x,y: Nat
  rew  eq(0,0)       = T
       eq(0,succ(x)) = F
       eq(succ(x),0) = F
       eq(succ(x),succ(y)) = eq(x,y)
  ```

  It is not allowed to use the same name more than once in an equation specification, as in the following example:

  ```
  var  x: Bool
       x,y: Nat
  ```

Equation-section The equation-section consists of the keyword **rew** followed by a series of equations:

```
rew data-term = data-term
```

- Both data-terms in an equation must be of the same sort.
- If the data-terms in an equation section contain variables, these variables must be declared immediately before that equation-section.
- The tools apply the equations as rewrite rules from left to right. However, this way of using the equations is not prescribed in the definition of μCRL itself.
- More information about the rewrite process and hints for writing equations can be found in Appendix III.

2.3 Specification of processes

A μCRL specification describes systems of communicating processes with data. Processes are viewed as sequences of atomic activities called 'actions'. Processes are represented by means of process-terms.

Data can be introduced in process specifications as parameters of actions and processes. A conditional (if-then-else construct) allows data to influence the course of a process.

Fig. 5 gives an overview of a process specification.

act	Actionname, Actionname ... : domain
proc	Processname (Varname: sortname, Varname: sortname ...) = process-term
comm	action-name | action-name = action-name
init	process-term

Figure 5: Process specification format

Actions Actions are represented by means of action-terms. Examples of action terms are `a`, `a(3)` and `a(T,F,3,f(g(x)))`.

Declaration of actions Actions are declared by means of the keyword **act**, followed by a (space separated) list of action declarations:

```
act Actionname-list : Domain
```

For example:

```
act a b c                        % simple actions
    read, write : Data           % read / write a datum
    send, receive : Data # NAT   % send / receive package consisting
                                 %   of a datum and a frame number
```

- The Actionname-list is comma separated.
- The domain is a list of declared sort names separated by hashes (#). This list defines the parameters of the action. For example, if an action `read` is declared as:

  ```
  act read: D # Bit
  ```

  two possible action-terms are: `read(d1,0)` and `read(d1,1)`.
- It is allowed to overload names of actions but the combination of the name and the sorts of its domain must be unique (except that a parameterless action and a sort may have the same name).

Predefined actions There are two predefined action names in μCRL:

`delta`	deadlock
`tau`	the internal action

Action-terms Action-terms consist of an action name possibly followed by a parameter-list (i.e. a comma separated list of data-terms between brackets). All action names must be declared and the sorts of their parameters must be the same as the sorts in the domain of the declaration.

Processes Processes are represented by means of process-terms. Process-terms describe the order in which the actions can happen. Process-terms consist of basic process-terms combined by means of operators.

Declaration of processes Processes are declared by means of the keyword **proc** followed by one or more process-declarations:

```
proc Processname (list of typed variables) = process-term
```

- It is allowed to overload names of processes but the combination of the name and the sorts of its variables must be unique (except that a parameterless process and a sort may have the same name).
- The list of typed variables is optional.
- The list of typed variables consists of one or more comma separated declarations of the form:

```
Varname : sortname
```

- The sortnames must be declared.
- Variables can not have the same name as function constants, parameterless actions or parameterless processes.
- A variable name may occur only once within each list.
- Recursion is allowed.

Some basic examples:

1. The first example declares a simple process, `X` that performs an `a` action followed by a `b` action:

```
proc X = a . b
```

2. The following declaration recursively defines a process that carries out infinitely many `a` actions:

```
proc X = a . X
```

3. The third example uses parameters to define a counter:

```
proc Count(n: Nat) = announce (n) . Count(succ(n))
```

Basic process-terms Basic process-terms are names of actions or processes possibly followed by a parameter list (i.e. a comma separated list of data-terms between brackets). All basic process-terms must be declared and the sorts of their parameters must be in accordance with their declaration.

Operators Fig 6 gives an overview of the process operators of μCRL.

- Action-lists are comma separated lists of action names.
- The priority of these operators: `@`, `.`, `<<`, `||`, `<| ... |>`, `+`.

The μCRL definition does not distinguish different kinds of operators and allows arbitrary nesting of all operators. For practical purposes it is useful to group them as in Fig 6:

- The operators `+`, `.`, `<| ... |>`, and `sum` are used in the **proc** section to specify the component processes of a possibly complex system.
- The operators `||`, `encap`, `hide`, and `rename` are used to glue these components together (usually in the **init** section). Be advised, that the lineariser is not able to linearise processes in which these operators occur within the scope of `+`, `.`, `<| ... |>`, and `sum`. If one tries to linearise such process the lineariser will produce the error message `Mixing pCRL with mCRL operators`.
- The time operators `<<` and `@` and the parallel operators `|_` and `|` are allowed by μCRL, but, except for the parser, the tool set does not handle processes in which these operators occur.

process-term `.` process-term	sequential composition
process-term `+` process-term	alternate composition
process-term `<|` boolean `|>` process-term	conditional
`sum` (variable, process term)	sum
process-term `||` process-term	parallel composition
`encap`({action-list}, process-term)	encapsulation
`hide` ({action-list}, process-term)	hide
`rename`({rename-list} process-term)	rename
process-term `<<` process term	time shift
process-term `@` data-term	at
process-term `|_` process-term	left merge
process-term `|` process-term	communication

Figure 6: Overview of process operators

Communication The keyword **comm** can be used to specify which actions can synchronise. A communication specification consists of the keyword **comm** followed by one or more declarations of the following form:

```
comm action-name | action-name = action-name
```

For example:

```
comm s2 | r2 = c2
```

- The names in the action-terms must be appropriately declared.
- The sorts of parameters of the three action terms must match.

The initial behaviour The initial behaviour of the system can be specified with the keyword **init** followed by a process-term:

```
init process-term
```

For example:

```
init Count(zero)
```

would cause the counter to count from zero onwards.

- The μCRL definition does not require a specification to have an **init** section, but the `instantiator` can not instantiate uninitialised state spaces.

3. The Tools

3.1 The tool: mcrl

The tool `mcrl` checks whether a specification is well-formed timed μCRL. In addition, it can transform certain μCRL specifications to a linear process operator format.

General description

Well-formedness check The tool `mcrl` checks whether a specification is well-formed (timed) μCRL as defined in [Gro97]. Roughly spoken, 'well-formed' means that the following conditions are satisfied:

- The specification is syntactically correct.
- All names in the specification (of sorts, functions, variables, actions, and processes) are appropriately declared. This means that there are sort names where there should be sort names, function names where these are required, and so on. It also means that every sort is declared only once, that there are no functions, actions and processes that have the same combination of name and domain sorts and that the names of variables are unique within their declaration and different from the names of constant functions, unparameterised actions and unparameterised processes.
- There are no empty sorts.
- The sort `Bool` is declared, as are the two constructors (`T` and `F`) of this sort.
- If the sort `Time` is declared, both `time0` and `le` are declared as functions of this sort.
- All data-terms conform with the declarations (i.e. they are type correct).
- Both data-terms of each equation are of the same sort.
- All conditions are of sort `Bool`.
- The term at the right-hand side of every `@` operator is of sort `Time`.
- If an action a is renamed to b, b is declared with respect to all the domains of a.
- The sorts of all communicating actions match.
- The communications are defined in such way that communication is associative and commutative.
- There is not more than one initial process declared.

Linearisation The tool `mcrl` can also be used to translate a well-formed μCRL specification to a linear process operator format[3] provided that the specification meets the following requirements:

- The process descriptions do not refer to time (i.e. neither the `@` nor the `<<` operator is used).
- The left merge (`||_`) and the communication merge (`|`) are not used to specify processes.
- Every process declaration must belong to one of the following syntactic categories:
  1. declarations in which action and process names are glued together by means of the operators `.`, `+`, `<| ... |>`, and `sum`
  2. declarations in which process names are glued together by means of the operators `||`, `hide`, `encap`, and `rename`.

  If this requirement is violated the lineariser will respond with the somewhat cryptic error message `Mixing pCRL with mCRL operators`.
- The operators `||`, `hide`, `encap` and `rename` are not used within the scope of the operators `.`, `+`, `<| ... |>`, and `sum`. If this requirement is violated the lineariser outputs the error message `parallel operator in the scope of pCRL operators`.

[3]Details of the linear process operator format and an impression of the linearisation algorithm are given in appendix II

- Recursion is guarded.[4]
- There is no recursion at the level of the `||`, `hide`, `encap` and `rename` operators.

The lineariser may need the following functions:

- the function `not` of type `Bool -> Bool`,
- the functions `and` and `or` of type `Bool#Bool -> Bool`,
- the function `eq` with target sort `Bool` for pairs of certain sorts.

If the lineariser arrives at a point where it needs one of these functions and that function is not declared it will produce an error message and exit subsequently. It is the responsibility of the author of the specification to provide appropriate rewriting rules for these functions. The rules for `not`, `and` and `or` should correspond to the logical operators with the same name (see for example Fig 4); `eq` is supposed to be a function that specifies of each pair of constructors of a certain sort whether they are the same or not. For example, for the sort `Bool` the rewrite section of `eq` might read:

```
rew eq(T,T) = T
    eq(T,F) = F
    eq(F,T) = F
    eq(F,F) = T
```

The lineariser does not handle terminating processes correctly. In most cases it exits with an appropriate error message if a process terminates, but it might also produce erroneous output without any warning. To avoid difficulties one should put a `delta` behind processes that terminate. This should not be done carelessly. Consider, for example the following process specification:

```
proc P = a.P.c + b
```

This specification specifies a process that executes zero or more `a` actions, followed by a `b` action, followed by as many `c` actions as there were `a` actions. One cannot simply put a `delta` behind the `c` and/or the `b`. One may not put a `delta` behind the `b` as this would result in a process with a different behaviour (no `c`'s will be performed). However, putting a `delta` behind the `c` but not behind the `b` will not solve the problem as this means that `P` may terminate by executing `b`. In many cases, the problem might be solved in the **init** section:

```
proc P = a.P.c + b
init P . delta
```

As this works only if there are no processes put in parallel[5] a better solution is reached by adding a process name:

```
proc X = P.delta
     P = a.P.c + b
```

Format `mcrl [ -linear | -tbfile | -stdout [-regular | -regular2] [-cluster [-binary] | -nocluster] ]` *infile* `| [-help] | [-version]`

Options

[4] The notion of guardedness is explained in Section II.2.3.

[5] if there are parallel processes, a `delta` in the init section will elicit the `Mixing pCRL with mCRL operators` error message

Output format If `mcrl` is invoked with a filename, *infile*, and without one of the output format options (`-linear`, `-tbfile`, `-stdout`) it will perform a well-formedness check on *infile*, and write the result to `stdout`.

If an output format option is specified `mcrl` will attempt to linearise the specification in *infile* after the well-formedness check (but only if the specification is indeed well-formed).

`-linear`	output is written in text format to *infile*`.lin`
`-tbfile`	output is written in tool bus format to *infile*`.tbf`
`-stdout`	output is written in tool bus format to `stdout`

Regularity When `mcrl` is invoked with the flag `-regular` or `-regular2` it attempts to generate an LPO by applying a regular linearisation method.[6] The tool does not check in advance whether such a translation exists. If such a translation does not exist `mcrl` will end up in a loop, and will ultimately crash due to lack of memory.

When `mcrl` is invoked without a regularity flag it will use stacks to linearise the specification.[7] Such a translation does always exist.

As specifications with stacks are difficult to understand or analyse, it is recommended to use regular linearisation right away. If the state space associated with a specification is finite, regular linearisation is always possible. This means that one needs stacks only to linearise specifications with an infinite state space. As the `instantiator` of the tool set is not yet able to instantiate infinite state spaces one might conclude that, currently, in practice, there will seldom or never be a reason to use stacks.

The difference between the flags `-regular` and `-regular2` is explained in Section II.6.3. It influences the way in which parameters of internally created process names are generated. Both methods have advantages and disadvantages. In practice one should try both methods to find out which one gives the best results. The use of `-regular` often leads to substantially less data parameters than the use of `-regular2`, which increases the speed of the lineariser and the instantiator. The `-regular2` flag is useful when there are a lot of similar structured process expressions in a specification. In this case `-regular2` might generate smaller state spaces than `-regular`. Finally, there are cases in which `-regular2` terminates and `-regular` not.

Clustering If a process term has several summands with the same action they might be packed together with the help of a sum operator. For example, the process term `a(T).X+a(F).X` can be written as the single summand `sum(b:Bool,a(b) . X)`. This is called *clustering*.

The cluster flags influence clustering during linearisation:

[none]	summands are clustered before putting two processes in parallel
`-nocluster`	summands are not clustered except in very simple cases
`-cluster`	summands are clustered before putting two processes in parallel and a second time at the end of linearisation processes.

Tool info	`-help`	provides a short description of the tool with all its flags
	`-version`	print version info

Error messages The tool `mcrl` may produce several kinds of error messages.

Command line errors These errors concern incompatible options such as `Options -tbfile and -linear cannot be used together` and the I/O errors `Cannot open file for output` (does the file exist in the relevant directory?) and `Cannot open file for input` (does there exist a file with the same name, of which you lack the privileges to change it?).

[6] See Section II.6 for an explanation.

[7] Details of this method are given in Section II.5.

Syntax errors After invocation `mcrl` will try to parse the specification. If it reaches a point where it does not know how to parse a certain string it exits immediately and produces the error message `line %n: parse error, near string %s`. This error message indicates the string that could not be parsed and the number of the line in which that string occurred. As is usual with parse errors, the error is always before the string mentioned in the error message and often not near the indicated line.

Other well-formedness errors Next, the tool `mcrl` checks non-syntactic well-formedness requirements (see Section 3.1). If it finds an error it exits immediately and tells you what's wrong. For example, if `mcrl` for the second time bumps on a declaration of sort `S` (which violates the requirement that sort names must be unique) it will stop and say that

`Sort 'S' appears more than once`.

Most error messages speak for themselves. Here are some hints regarding the more difficult ones.

- The wording of the messages is not always consistent. For example, if `mcrl` for the second time bumps on a declaration of a certain function name, it will say something like

  `Function 'f' appears twice`

  (even if `f` is declared four times!).

- Some error messages refer to the order in which the well-formedness is checked. This order is not necessarily the same as the order in which the terms appear in the specification. For example, if a specification states:

  ```
  act a
  sort A
  func a:->A
  ```

  `mcrl` will complain that

  `Action name 'a' is already in use`,

  despite the fact that in the specification `act a` appears before `func a`.

- The complaint that a certain function is 'badly' or 'incorrectly' used may indicate:

  1. that the checker has bumped at an undeclared name at a position where it expects the name of a function *or a variable*, or
  2. that one of the parameters of the function about which the checker complains is of the wrong type

  For example, if `f` is not declared, the process declaration

  ```
  proc X(b1: Bool) = sum(b2: Bool, a . X(f(b1,b2)))
  ```

  will evoke the error message:

  ```
  The function f in action or process X(f(b1,b2)) in X is
  incorrectly used
  ```

  The same message will be produced if `f` is declared, but if its actual parameters are of the wrong type. For example, if `f` is declared as:

  ```
  map f: Nat # Nat -> Bool
  ```

On the other hand if the name of one of the actual parameters of `f` is undeclared, `mcrl` will complain about that name. For example, if `b2` is not declared, the declaration

```
proc X(b1: Bool) = a . X(f(b1,b2))
```

will evoke the complaint:

`The function b2 in action or process X(f(b1,b2)) in X is incorrectly used.`

- If the checker complains that a certain variable is 'already declared as function/variable' this means that the name of that variable is also declared as the name of a (constant) function, another variable in the same declaration, or of an (unparameterised) action or process.

  For example, the following declaration

  ```
  act a b
  proc X(b: Bit) = a . X (invert(b))
  ```

  will provoke the error message

  `Variable 'b' is already declared as function/variable.`

  because `b` is declared both as a variable and as an action.

Memory problems The well-formedness checker of `mcrl` attempts to avoid core dumps. This means that it will exit with an error message if there are memory problems. Examples of this kind of error messages are:

```
Too many functions while storing function '%s'

Out of memory while storing function '%s'
```

The first message indicates that in an internally created table there is no room left to store the name of a function; the second message indicates that `mcrl` has not enough memory to store the function.

Error messages of the lineariser If one of the output flags is specified `mcrl` will check whether or not the specification meets the requirements for linearisation (see Section 3.1). This check takes place after the well-formedness check. If `mcrl` discovers a violation it will exit immediately with an appropriate message. If the requirements are met it will attempt to linearise the specification. The lineariser may need the functions `eq`, `and`, `or`, or `not` of sort `Bool`. If the lineariser arrives at a point where it needs one of these functions and that function is not declared it will produce an error message and exit immediately.

Known problems

1. Processes that terminate are not linearised correctly, but a warning is not always given. This does not apply to intermediate termination, where one process terminates and another continues. The general solution is to put a `delta` behind processes to prevent them from terminating.

2. For the enumerated types that are generated when clustering, there are no equality functions `eq` defined. This means that linearising an already linearised system may fail due the absence of these functions.

3. It is not checked whether the functions `and`, `or`, `not` and `eq` that are sometimes required in the process are defined in accordance with the normal interpretation of these functions. So, it would not be noticed if `not` would be defined by `not(T)=T` and `not(F)=F`. The result of the lineariser depends on the correct definition, and becomes wrong in this case.

4. When mcrl is invoked with the -regular or -regular2 flag, and the control of the process is not finite state, then mcrl will loop, and crash after a while due to lack of memory.

5. The lineariser has not been optimised. It uses the general, but not extremely efficient, level 1 aterm functions (with some exceptions). Furthermore, its internal data structures are rather basic. This may explain that for very large systems, or systems with a lot of parallelism it may take some time to finish linearisation.

3.2 The tool: msim

The tool msim can be used to single step through a process described in μCRL (text or .tbf format).

Overview A typical session with msim consist of the following steps:

1. start msim by typing msim file.tbf (Fig. 7),
2. initiate the simulation by clicking the Start button (Fig. 8),
3. continue the simulation by selecting actions in the Menu display (Fig. 9),
4. finish the simulation by clicking the Quit button.

Start msim The simulator msim is invoked by:

msim [*file*]

The simulator will present a window that allows interactive simulation of the system described in the argument *file* (Fig 7).

The argument *file* must be a well-formed μCRL specification in text format or an LPO in .tbf format. If the argument *file* is omitted msim will simulate the process described in share/abp.[8]

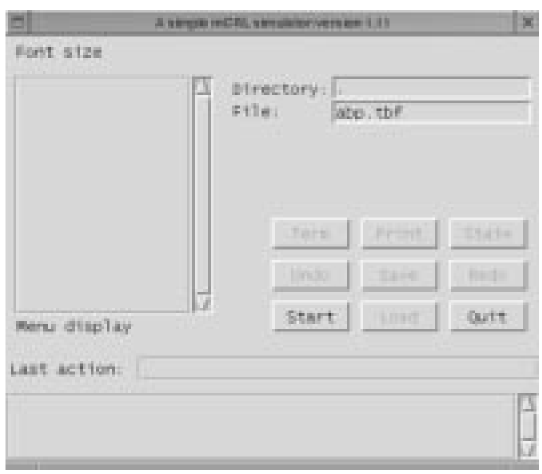

Figure 7: The msim window - immediately after start up

Initiate the simulation To start the simulation click the Start button. The simulator will translate the data types to C code and load them in the program. If all goes well the simulator presents a list of actions in the Menu display window (see Fig 8). Otherwise an error message will appear in the message window (bottom of the screen).

[8]This path is relative to the mcrl top directory

If the argument *file* is in text format `msim` will first generate a `.tbf` file. It does so by automatically invoking the command `mcrl -tbfile` *file*. Note, that the default method of linearisation is used. In order to get regular linearisation (the state description of which is much more easy to interpret) one must linearise the specification before invoking `msim` (`mcrl -tbfile -regular` *file*) and one must invoke `msim` with the `.tbf` file (`mcrl` *`file`*`.tbf`).

Figure 8: The `msim` main window after pressing start

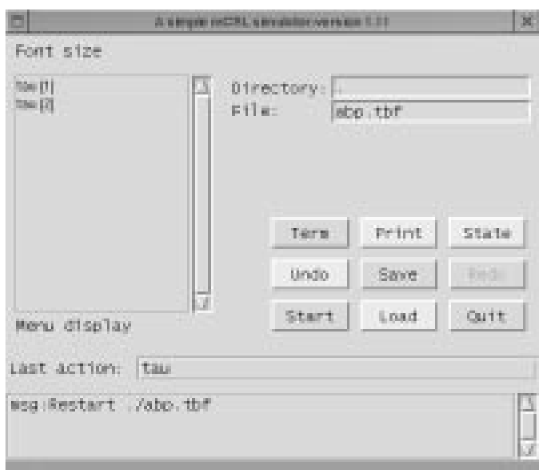

Figure 9: The `msim` main window

Continue the simulation The `msim` main window (Fig. 9) offers the following possibilities (see also table 1):

- You can single step through the system by selecting actions in the `menu display`.
- The program maintains a trace that can be saved in text(`Print`) or binary format (`Save`).

Start	Return to the initial state
State	Display the state vector describing the current state
Term	Display the value of a function in the current state
Undo	Back to the previous state in the trace
Redo	Forward to the next state in the trace
Print	Save the trace in text format
Save	Save the trace in binary format
Load	Load a trace in binary format
Quit	Quit msim

Table 1: The buttons of the `msim` main window

- You may return to the previous state in the trace by clicking the `Undo` button. It is possible to undo the whole sequence of chosen actions up to the initial state.
- You may redo undone actions by clicking `Redo`.
- To display a description of the current state click `State` (the interpretation of this description requires both experience and knowledge of the linearisation process).
- To display the value of a certain function click `Term`. A term window pops up (Fig 10). Enter the function and click `OK`. It is possible to display more than one term window.
- You may change the size of the fonts in the `Menu display` by clicking on `Font size`

Figure 10: A term window

Finish and Restart

- To finish the simulation click `Quit`.
- To restart the simulation click `Start`.
- To restart the simulation with another specification, change the `File` and /or `Directory` and click `Start`.
- To load a saved trace in binary format click `Load`.

References The tool `msim` is described in [Kor].

Known problems There are no known problems with `msim`.

3.3 The tool: instantiator

The tool `instantiator` reads a file in aterm internal format (a `.tbf` file) and generates a transition system. By default, the generated transition system is in AUT format. This transition system can be read and manipulated by several tools among which the CÆSAR ALDÉBARAN Development Package. The instantiator can also produce transition systems in SVC format [Lan01]. This is a new, compact format, the tool support of which is rather limit at the moment.

General description

The `.aut` format A `.aut` file is a text file that describes a state space by means of a list of transitions. Each transition consists of the number of the starting state, the name of an action, and the number of the resulting state (in that order). The first line in the `.aut` file describes the state space as a whole (initial state, number of transitions, number of states). Here is an example of a very simple `.aut` file which describes the space pictured in Fig. 11:

```
des (0,5,4)
(0,"a",1)
(0,"b",2)
(1,"c",3)
(2,"c",3)
(3,"d",0)
```

State space generation The `instantiator` reads a file with a linear process in aterm internal format and explores the state space. The exploration starts with the initial state (which must specified by means of an **init** clause). The `instantiator` traverses all summands of the linear process operator. If the condition of a summand applies, a transition is added to the `.aut` file. This transition consists of the number of the state being explored, the action label of the summand, and the number of the resulting state. The initial state has label zero. If a resulting state occurs for the first time it gets the next free number. The `instantiator` lists all resulting states for further exploration. The initial state plus all known resulting states (known at a certain point in the exploration process) together are called 'visited states'. If a state is completely explored the instantiator proceeds to the next state on the list (i.e. to the next visited state), until all states are explored.

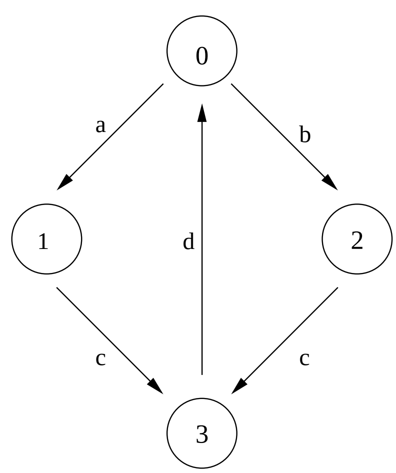

Figure 11: A simple state space

Format `instantiator [-svc | -svc-num] [-i] [-nohash | -simple] [-case | -order` *`label`*`]` `[-rewr] [-monitor] [-deadlock]` *`infile`* `| -help | -version`

Options

Overview

`-svc`	output in SVC format; states labeled with state vectors
`-svc-num`	output in SVC format; states labeled with numbers
`-i`	print the internal action as `i` instead of `"tau"`
`-nohash`	don't use a hash table during rewriting
`-simple`	another instantiator is run which does not use a hash table during rewriting,
`-case`	start with the superficial variables when evaluating sum variables in conditions (useful after applying `structelm`),
`-order`	specify the order in which sum variables in conditions are evaluated (for experimental use only),
`-monitor`	print to `stdout` how many states have been explored
`-deadlock`	print the deadlock states to a `.dlk` file (no `.aut` file generated)
`-help`	print a short description of the tool and its flags
`-version`	print version info

Output format By default `instantiator` outputs a `.aut` file. If the `-svc` or `-svc-num` flag is enabled, the output is in SVC format. In the first case (`-svc`) the states are labeled with aterms, specifying the state vector; in the second case (`-svc-num`) the states are labeled with numbers.

The CÆSAR ALDÉBARAN tool set assumes that the internal action is denoted by `i`, whereas `instantiator` by default produces `"tau"`. The `-i` flag causes `instantiator` to produce `i`. This flag is necessary if one wants to reduce the `.aut` file modulo branching bisimulation or modulo weak bisimulation with CÆSAR ALDÉBARAN.

Hash table during rewriting By default the `instantiator` checks during each rewriting step if the `-no-hash` flag is set. If this flag is not set then a hash table containing normalised terms is present and the rewriting steps use this hash table. If the `-no-hash` flag is set then the rewriting steps don't use a hash table. The repeated checking of the flag consumes time. This is the reason why the `-simple` flag is introduced. If this flag is set another piece of code is run in which there are no checks and which does not use a hash table during rewriting. This instantiator runs faster on small specifications in which the benefits of hash tables do not outweight the disadvantages of an extra check.

Enumeration order As is explained in the appendix (Section III.3) the `instantiator` has a component, the `enumerator`, which determines which instances of a combination of sum variables satisfy a condition. The speed of this component depends to a large extend on the order in which the different sum variables are evaluated. This order is affected by the `-case` and `-order` flags.

By default, sum variables of enumerated types[9] are evaluated before other variables and within each class the variables are evaluated in the order of the frequency with which they occur in the condition.

After applying `structelm` followed by `rewrite -case`, the conditions contain many case functions[10] and there are no case functions within non-case functions
(e.g. `C(e-1,C(e-2,C(e-3,f(i),g(j)),C(e-4),h(k),h(l)), C(e-5,f(k),g(l)))`). In this case it is faster to start with the most superficial variables. This is accomplished by the `-case` flag.

It is also possible to specify the order in which sum variables are evaluated by means of the `-order` flag. This flag is for experimental use only. Possible orders are: `no`, `enum_big_freq`, `big_freq`, and `min_depth`:

[9] *Enumerated types* are sorts that have only constants as constructors

[10] The term 'case function' is explained in Section 3.9.

`-order no`	the sum variables are evaluated in the order in which they appear in the conditions
`-order enum_big_freq`	the sum variables of enumerated types are evaluated before other variables and within each class the variables are evaluated in the order of the frequency with which they occur in the condition (this is the same as the default order)
`-order big_freq`	the sum variables are evaluated in the order of the frequency with which they occur in the condition
`-order min_depth`	the sum variables are evaluated in the order of their depth (from superficial to deep) (this is equivalent to `-case`)

The `-prerewr` *flag* If the `-prerewr` flag is enabled the conditions are reduced (by applying the rules specified in the **rew** section), before they are submitted to the enumerator. This speeds up the enumeration process considerably. However, it is recommended that a specification is rewritten before it is instantiated and in this case the `-prerewr` flag is superfluous.

Dealing with large state spaces The `.aut` format is rather inefficient and may grow very large. The `-monitor` and `-deadlock` options assist in dealing with large state spaces.

The `-monitor` flag prints every 1000 explored states a message to stdout. This message says how many states have been explored, how many states have been visited, and how many transitions there currently are in state space.

If the `-deadlock` flag is enabled `instantiator` explores the state space but does not generate a `.aut` file. Instead it prints the deadlock states to a `.dlk` file.

Tool info The `-help` flag provides a short description of the tool, and the `-version` flag described its version.

Known problems

1. The current size of state spaces is restricted to approximately $3 * 10^7$ states, but this figure depends highly on the process for which the state space is generated. Such a large state space needs approximately 1.5 Gb of memory.

2. You get problems if you try to instantiate several files in the same directory at the same time or if you run `msim`, `instantiator`, `rewr`, `constelm`, and/or `parelm` at the same time on files residing in one directory. These tools produce produce and use two files called `AUXTERM.c` and `REWRITERALT.c` which contain C-code for reducing data terms. The problems occur because one instantiation will overwrite (or use) the code of another instantiation.

References The instantiator is described in [DG95].

3.4 The tool: pp

The tool `pp` pretty prints a linear process operator in `.tbf` format.

General description The pretty printer `pp` reads an LPO in `.tbf` format and transforms it to ascii format. Output is written to `stdout`.

Format `pp [-ascii] [`*infile*`] | [-help] | [-version]`

Options

- By default `pp` outputs a format similar to that of the tool `mcrl` with the `-linear` option. This output is human readable and can also serve as input for the tool `mcrl`.
- If the `-ascii` flag is used, `pp` outputs the ascii representation of the aterm in *infile*. This can be useful to transform a binary represented aterm (`.tbf` format) to a slightly more readable form.
- If *infile* is omitted, `pp` reads from `stdin`.
- `-help` provides a short description of the tool and its flags.
- `-version` prints version info

Known problems There are no know problems with the pretty printer, `pp`.

3.5 The tool: rewr

The filter `rewr` normalises an LPO.

General description This filter reduces an LPO in `.tbf` format by applying the equations of the data types. A summand is removed if its condition turns out to be false. Output is written to `stdout`.

Format `rewr [-ascii] [-hash] [-case] [`*infile*`] | [-help] | [-version]`

Options

- By default `rewrites` outputs binary aterm format (`.tbf` format).
- If the `-ascii` flag is used, the output is an aterm in ascii representation.
- If the `-hash` flag is used, `rewr` uses a hash table for storing normalised terms.
- If the `-case` flag is used, `rewr` uses implicit rewrite rules handling case functions. These rules pulls the case functions outwards. The command `rewr -case` must be used after the command `structelm`, because `structelm` generates a lot of (new) case functions. Information about case functions and their implicite rewrite rules can be found in [LG01].
- If *infile* is omitted, `rewr` reads from `stdin`.
- `-help` provides a short description of the tool and its flags.
- `-version` prints version info

Known problems See item 2 of Section 3.3 for the only known problem with `rewr`.

3.6 The tool: parelm

The tool `parelm` removes from an LPO those data parameters and sum variables that do not influence the behaviour of the system. This may reduce the state space considerably.

General description The filter `parelm` reads an LPO in `.tbf` format and removes the data parameters and sum variables that do not influence the behaviour of the system. For example, if process `P` is defined as

```
P(x: Nat) = a . P (succ(x))
```

x is superfluous and will be removed by `parelm`.

The tool carries out the following algorithm:

1. Select the parameters that occur in the conditions and/or in the action arguments of the LPO,
2. Extend the selection with all parameters that depend on parameters which are already selected,
3. Repeat step 2 until no parameters are added to the selection,
4. Remove all parameters that are not selected,
5. For each summand: remove all the sum variables that occur neither in the condition, nor in the action argument, nor in the arguments of the new state.

The output is written to `stdout`.

Format `parelm [-ascii] [`*infile*`] | [-help] | [-version]`

Options

- By default the output is in binary aterm format (`.tbf`)
- If the `-ascii` flag is used, the output is an aterm in ascii representation.
- If *infile* is omitted, `parelm` reads from `stdin`.
- `-help` provides a short description of the tool and its flags.
- `-version` prints version info

Known problems There are no known problems with `parelm`.

3.7 The tool: constelm

The filter `constelm` removes from an LPO those data parameters that are constant throughout any run of the process. This may speed up state space generation considerably. It may also improve the readability of the LPO.

General description The tool `constelm` reads an LPO in `.tbf` format, looks for data parameters that are constant throughout any run of the process and replaces them by this constant value. For example, in the following specification the value of y is constant

```
proc P(x: Nat, y: Nat) = a(x,y) . P(succ(x),3)
init P(x,3)
```

and `constelm` will replace y by `3`. Output is written to `stdout`.

Format `constelm [-ascii] [`*infile*`] | [-help] | [-version]`

Options

- By default `constelm` outputs binary aterm format (`.tbf`).
- If the `-ascii` flag is used, the output is an aterm in ascii representation.
- If *infile* is omitted, `constelm` reads from `stdin`.
- `-help` provides a short description of the tool and its flags.
- `-version` prints version info

Known problems There are no known problems with `constelm`.

3.8 The tool: structelm

The tool `structelm` replaces terms with a constructor function as head symbol by the name of the constructor and its arguments. In this way variables occurring in subterms are better exposed and more amenable to be eliminated by one of the tools.

General description The filter `structelm -expand` *sortname* reads an LPO in `.tbf` format and replaces terms of sort *sortname* with a constructor function as head symbol by the name of the constructor and its arguments. For instance, a list expression $in(v, l)$ is split into the value *in*, and terms v and l, and a list expression *nil* is replaced by the value *nil*. Consequently, a variable of sort *List* is replaced by three variables. The first one to indicate whether the head symbol of the term represented by the variable starts with *in* or *nil* and two to represents the two arguments in case the term starts with *in*. This expansion method is called `structelm`, short for structure elimination. The application of `structelm` itself does not optimise a specification, the tool is useful only in combination with `parelm` and/or `constelm`. The latter tools might find more parameters that can be eliminated or replaced by constants if the sorts in the summands are first split up by `structelm`. Output is written to `stdout`.

Format `structelm [-expand` *sortname*`] [-depth` *number*`] [-torewr] [-binary] [-ascii] [`*infile*`] | [-report] | [-help] | [-version]`

Options

What to expand

- The `-expand` flag allows you to specify which sorts must be expanded. If the flag `-expand` is omitted then all sorts are taken which have at least one constructor with at least one argument.
- The `-depth` flag can be used to specify the depth of the expansion.

Method

- The `-binary` flag translates the name of the constructor which is considered by `structelm` as a value into a row of binary values. An advantage of the use of this flag is a more compact storage of terms.

Input and output format

- By default `structelm` outputs binary aterm format (`.tbf`).
- If the `-ascii` flag is used, the output is an aterm in ascii representation.
- If *infile* is omitted, `structelm` reads from `stdin`.

Info

- `-report` returns a list of composite sorts
- `-help` provides a short description of the tool and its flags.
- `-version` prints version info

Known problems There are no known problems with `structelm`.

References `structelm` is described in [LG01].

3.9 The tool: sumelm

The filter `sumelm` simplifies an LPO by replacing sum variables that must be equal to a certain data term with that data term. This might improve the speed of the `instantiator` and the effect of `constelm` and `parelm`.

General description The tool `sumelm` looks for summands that have a condition which states that a sum variable (i.e. a variable bound by the sum operator) must be equal to some data term and replaces that variable by that data term. An example is the following summand of the process `P(e:Nat)`:

```
sum(d:D, sum(f:Nat, read(d,f) . Q(d,e)  <| eq(f,e)|> delta))
```

The tool `sumelm` will replace this by

```
sum(s:D, read(d,e) . Q(d,e))
```

The tool will replace sum variables that occur in conditions consisting of a single `eq` function as well as in conditions in which the `eq` function occurs directly within one or more `and`, `or`, or case functions. So, if `d` is a sum variable, it will be replaced if the condition is `and(eq(d,e), eq(f,g))`, but not if the condition is `fancyboolean(eq(d,e), eq(f,g))`.

A certain function (say `f`) of a certain sort (say `S`) is a case function if it satisfies the following conditions:

- in regard to its signature
  - The first argument is of an enumerated sort (i.e. a sort with a finite number of constructors that have no arguments), say `E` with n constructors.
  - This first argument is followed by n arguments of sort `S`.
- in regard to its rewriting system:
  - There is an equation of the form

    $\mathrm{f}(\mathrm{c}_i,\mathrm{v}_1 \ldots\ldots \mathrm{v}_n) = \mathrm{v}_j$

    in which
    - c_i is a constructor of `E`
    - $\mathrm{v}_1 \ldots\ldots \mathrm{v}_n$ are different variables of sort `S`
    - v_j is one of the variables $\mathrm{v}_1 \ldots\ldots \mathrm{v}_n$

    for each constructor of `E`.
  - The right hand side of all these equations differ from each other.

Output is written to `stdout`.

The application of `sumelm` can be useful for two reasons:

- It speeds up the `instantiator` because this tool need not try all possible values of the eliminated sum variable.
- After application `sumelm` to an LPO tools such as `constelm` and `parelm` might do a better job.

Format `sumelm [-ascii] [`*infile*`] | [-help] | [-version]`

Options

- By default `sumelm` outputs binary aterm format (`.tbf`).
- If the `-ascii` flag is used, the output is an aterm in ascii representation.
- If *infile* is omitted, `sumelm` reads from `stdin`.
- `-help` provides a short description of the tool and its flags.
- `-version` prints version info.

Known problems There are no known problems with `sumelm`.

References

[BFG+01] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 250–254, 2001.

[BG93] M.A. Bezem and J.F. Groote. A formal verification of the alternating bit protocol in the calculus of constructions. Technical Report 88, Utrecht University, Logic Group Preprint Series, 1993. ftp://ftp.phil.uu.nl/pub/logic/PREPRINTS/preprint88.ps.Z.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[DG95] D. Dams and J.F. Groote. Specification and implementation of components of a μCRL toolbox. Technical Report 152, Utrecht University, Logic Group Preprint Series, December 1995. ftp://ftp.phil.uu.nl/pub/logic/PREPRINTS/preprint152.ps.Z.

[Fok00] W. J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. Springer Verlag, 2000.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer Verlag, 1995.

[GPU01] J. F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, (48):39–70, 2001.

[Gro97] J.F. Groote. The syntax and semantics of timed μCRL. Technical Report SEN-R9709, CWI, Amsterdam, 1997. ftp://ftp.cwi.nl/pub/CWIreports/SEN/SEN-R9709.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf – reference manual –. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[Kor] H.P. Korver. Building a simulator in the μCRL toolbox. Technical Report CS-R9632, CWI, Amsterdam.

[Lan01] I.A. van Langevelde. A compact file format for labelled transition systems. Technical Report SEN-R0102, CWI, Amsterdam, 2001.

[LEW96] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley/Teubner, 1996.

[LG01] B. Lisser and J.F. Groote. Computer assisted manipulation of algebraic process specifications. Technical Report SEN-R0117, 2001.

[MM94] S. Mauw and J.C. Mulder. Regularity of BPA-systems is decidable. In B. Jonsson and J. Parrow, editors, *Proc. CONCUR '94*, volume 836 of *Lecture Notes in Computer Science*, pages 34–47. Springer Verlag, 1994.

Appendix I
The syntax of timed μCRL

In this section the syntax of timed μCRL specifications is given. It is defined in the Syntax Definition Formalism (SDF) [HHKR89]. According to the convention in SDF we write syntactical categories with a capital, and keywords with small letters. The first LAYOUT rule says that spaces (' '), tabs (`\t`) and newlines (`\n`) may be used to generate some attractive layout and are not part of the μCRL specification itself. The second LAYOUT rule says that lines starting with a `%`-sign followed by zero or more non-newline characters (`~[\n]*`) followed by a newline (`\n`) must be taken as comments and are therefore also not a part of the μCRL syntax.

A `Name` is an arbitrary string over `a-z`, `A-Z`, `0-9` and the special characters `^_'-`. By a default SDF convention keywords cannot be a `Name` at the same time. In the context free syntax most items are self-explanatory. The symbol `+` stands for one or more and `*` for zero or more occurrences. For instance { `Name ","`}`+` is a list of one or more `Name` separated by commas, without a trailing comma.

The phrase {`right`} means that an operator is right-associative and {`assoc`} means that an operator is associative. The phrase {`bracket`} says that the defined construct is not an operator, but just a way to disambiguate the construction of a syntax tree.

The priorities say that the operator '`@`' has highest and `+` has low est priority when parsing process terms with ambiguous bracketing.

```
exports
 sorts Name
       Name-list
       Domain
       Sort-specification
       Function-specification
       Function-declaration
       Equation-specification
       Variable-declaration
       Variables
       Data-term
       Equation-section
       Single-equation
       Process-term
       Renaming
```

```
      Variable
      Process-specification
      Process-declaration
      Action-specification
      Action-declaration
      Communication-specification
      Communication-declaration
      Initial-declaration
      Specification
lexical syntax
      [ \t\n]                             -> LAYOUT
      "%" ~[\n]* "\n"                     -> LAYOUT
      [a-zA-Z0-9^_'\-]+                   -> Name
context-free syntax
      { Name ","}+                        -> Name-list
      { Name "#"}+                        -> Domain
      "sort" Name+                        -> Sort-specification
      "func" Function-declaration+        -> Function-specification
      "map" Function-declaration+         -> Function-specification
      Name-list ":" Domain "->" Name      -> Function-declaration
      Name-list ":" "->" Name             -> Function-declaration

      Variable-declaration
                 Equation-section         -> Equation-specification
      "var" Variables+                    -> Variable-declaration
                                          -> Variable-declaration
      Name-list ":" Name                  -> Variables
      Name                                -> Data-term
      Name "(" { Data-term "," }+ ")"     -> Data-term
      "rew" Single-equation+              -> Equation-section
      Data-term "=" Data-term             -> Single-equation
      Process-term "+" Process-term       -> Process-term {right}
      Process-term "||" Process-term      -> Process-term {right}
      Process-term "||_" Process-term     -> Process-term
      Process-term "|"   Process-term     -> Process-term {right}
      Process-term "<|" Data-term "|>"
                 Process-term             -> Process-term
      Process-term "." Process-term       -> Process-term {right}
      Process-term "@" Data-term          -> Process-term
      Process-term "<<" Process-term      -> Process-term {left}
      "delta"                             -> Process-term
      "tau"                               -> Process-term
      "encap" "(" "{" Name-list "}" ","
                     Process-term ")"     -> Process-term
      "hide" "(" "{" Name-list "}" ","
                     Process-term ")"     -> Process-term
      "rename" "(" "{" { Renaming "," }+
                 "}" "," Process-term ")" -> Process-term
      "sum" "(" Variable ","
                     Process-term ")"     -> Process-term
      Name "(" { Data-term "," }+ ")"     -> Process-term
      Name                                -> Process-term
      "(" Process-term ")"                -> Process-term {bracket}
      Name "->" Name                      -> Renaming
      Name ":" Name                       -> Variable
```

```
      "proc" Process-declaration+          -> Process-specification
      Name "(" { Variable "," }+ ")" "="
                            Process-term   -> Process-declaration
      Name "=" Process-term                -> Process-declaration
      "act" Action-declaration+            -> Action-specification
      Name-list ":" Domain                 -> Action-declaration
      Name                                 -> Action-declaration
      "comm" Communication-declaration+    -> Communication-specification
      Name "|" Name "=" Name               -> Communication-declaration
      "init" Process-term                  -> Initial-declaration

      Sort-specification                   -> Specification
      Function-specification               -> Specification
      Equation-specification               -> Specification
      Action-specification                 -> Specification
      Communication-specification          -> Specification
      Process-specification                -> Specification
      Initial-declaration                  -> Specification
      Specification Specification          -> Specification {assoc}

priorities
      Process-term "@" Data-term -> Process-term >
      Process-term "." Process-term -> Process-term >
      Process-term "<<" Process-term -> Process-term >
      { Process-term "||" Process-term -> Process-term,
        Process-term "|" Process-term -> Process-term,
        Process-term "||_" Process-term -> Process-term } >
      Process-term "<|" Data-term "|>" Process-term -> Process-term >
      Process-term "+" Process-term -> Process-term
```

Appendix II
Linearisation

This chapter describes in an informal style the linear process operator format (LPO) and the linearisation process as it is implemented in the tool set. The tool set implements three methods for linearisation: the default method and two regular methods. A formal treatment of the default method of linearisation is given in [GPU01]. There is no formal treatment of regular linearisation in μCRL yet.

The first section (II.1) of this chapter describes the linear process operator format (i.e. the output of the linearisation process). The second section (II.2 discusses the required input of the linearisation process. The remaining sections (II.3– II.7) discuss the linearisation process itself.

This chapter requires some experience with specification in μCRL.

II.1. The linear process operator format (LPO)

An LPO describes a system as a process that consists of a series of summands; each summand gives a condition, an action that can be performed if that condition applies, and the modified process that results from that action.

As an example, consider a simple buffer that can receive a datum with the action `read` and deliver it with the action `send`. The following specification of that buffer is not linear, because there are two actions before the recursive call of the process `Buffer`:

```
proc Buffer = sum(d:Datum, read(d).send(d).Buffer)
```

The specification can be linearised by dividing the `Buffer` process in two phases. In phase 1 a datum is received, in phase 2 the datum is delivered. The linearised `Buffer` needs two parameters: one to keep track of a read datum and one to keep track of the phase.

```
proc Buffer(d1:Datum, phase:Phase) =
   sum (d2:Datum, read(d2).Buffer(d2,s)  <| eq(phase,r) |> delta
   + send (d1).Buffer(d1,r)              <| eq(phase,s) |> delta
```

II.2. The input of the linearisation process

II.2.1 Overview of requirements

As said in Section 3.1, the lineariser of the tool set requires as input a μCRL specification that meets the following criteria:

- There are no references to time.
- Every process declaration belongs to one of the following syntactic categories:
  1. declarations in which action and process names are glued together by means of the operators `.`, `+`, `<| ... |>`, and `sum`
  2. declarations in which process names are glued together by means of the operators `||`, `hide`, `encap`, and `rename`.
- The process names occurring in a declaration of the first type are declared by means of a declaration of the first type.[1]
- Process names declared by a declaration of the second type are not used recursively.
- Recursion is guarded.

The first requirement (no time operators) speaks for itself. The next three requirements amount to the requirement that pCRL and parallel pCRL are neatly separated. This is discussed in the next subsection (II.2.2). The last requirement (guarded recursion) is discussed in subsection II.2.3.

II.2.2 Separating pCRL *from parallel* pCRL

Due to the second requirement, the set of process declarations to be linearised can be divided into two parts. These parts are treated differently during linearisation. These parts are called the *p*CRL and the *parallel p*CRL part:[2]

- The pCRL part consists of the declarations in which action and process names are glued together by means of the operators `.`, `+`, `<| ... |>`, and `sum`. All process names occurring in the pCRL part must be declared in the pCRL part. Recursion is allowed (provided it is guarded).
- The parallel pCRL part consists of the declarations in which process names are glued together by means of the operators `||`, `hide`, `encap`, and `rename`. The process names occurring in this part are defined in either the pCRL part or in this part. Recursion is not allowed.

```
proc S(c:Nat) = sum(d:D, accept(d).send(d,c).S(succ(c)))
     R(e:Nat) = sum(d:D, read(d,e).deliver(d).R(succ(n)))
     System(c:Nat, e:Nat) = S(c) || R(e)
```

Figure II.1: A simple Sender/Receiver system

Consider for example the process specification of fig. II.1. This specification specifies a system consisting of two processes in parallel:

- a sender (`S`) which accepts a datum from an application, adds a frame number and sends it to the network
- a receiver (`R`) which reads a datum from the network (if it has the appropriate frame number) and, sends it to an application

The declarations of `S` and `R` together form the pCRL part, the declaration of `System` forms the parallel pCRL part.

[1] in other words: the operators `||`, `hide`, `encap` and `rename` must not be used within the scope of the operators `.`, `+`, `<| ... |>`, and `sum`.

[2] pCRL is short for pico CRL, a subset of μCRL (micro CRL).

II.2.3 Guardedness
A process name, `X`, *depends* on another process name, `Y`, if `Y` is used to declare `X`. For example, if `X` is declared as `X = a.Y + Z` then `X` depends on `Y` and `Z`.

An occurrence of a process name is *guarded* if there is an action before that occurrence in the sequence that contains that occurrence. For example, `X` and `Y` are guarded in `X = a.X.Y + Z` (by the action `a` before `X` and `Y` in the sequence `a.X.Y`) but `Z` not (there is no action before `Z` in the sequence `Z`).

The lineariser requires that recursion is guarded in the sense that there are no cyclic unguarded dependencies in the specification.[3] For example, the following specification will be not be linearised:

```
proc X = a.X + Y
     Y = b.Y + Z
     Z = c.Z + X
```

because of the cycle: `X` depends on an unguarded `Y`, `Y` depends on an unguarded `Z`, `Z` depends on an unguarded `X`. Similarly,

```
proc X = X <| F |> a.Y
     Y = a.Y
```

will not be linearised, because `X` is unguarded in `X`. An example of a specification that will be linearised is:

```
proc X = Y
     Y = a.X
```

`X` depends on an unguarded `Y` but `Y` is completely guarded (i.e. `Y` has no unguarded dependencies).

II.3. Overview of the linearisation process

The linearisation process consists of two main steps:

1. The pCRL part is transformed into a single LPO, and the parallel pCRL part is appropriately updated.
2. The two parts are integrated into one single LPO.

The tool set provides three methods for performing the first main step:

- A general method
- Two regular methods, described in Section II.6

The general method applies to all specifications that satisfy the requirements. The regular methods do not apply in all cases.

Both the general method and the regular methods have an important sub step, namely combining processes. This sub step is introduced in Section II.4. Section II.5 describes the general method (for the first main step), Section II.6 the regular methods (for the first main step). The second main step is discussed in Section II.7.

II.4. Combining processes

The two main steps in linearisation can each be divided in several sub steps. One important sub step (both in general and in regular linearisation) is the combination of two or more processes into one

[3] Note that this notion of 'guarded recursion' differs slightly from the one used in [Fok00]

process without changing the system. This can be done by means of a state parameter that indicates which of the original processes the combined process should execute.

As an example of this sub step, consider the two processes `S` and `P` of fig. II.1. These two processes can be combined into one process (`Combined`) without changing the system, provided that the declaration of `System` is properly updated (see fig. II.2). In order to achieve this, a new sort `State` is introduced. The constructors of this sort are `S` and `P`. A parameter of this sort is introduced in the process `Combined` to indicate whether `Combined` must behave as the original `S` process or as the original `R` process.

```
proc Combined(X:State, c:Nat, e:Nat) =
         sum(d:D, accept(d).send(d,c).Combined(S,succ(c),e))
             <| eq(X,S) |>  delta
         + sum(d:D, read(d,e).deliver(d).Combined(R,c,succ(e)))
             <| eq(X,R) |> delta
      System(c:Nat, e:Nat) = Combined(S,c,e) || Combined(R,c,e)
```

Figure II.2: Combined process that acts as a Sender or as a Receiver depending on the State-parameter

The lineariser of the tool set represent states by means of numbers in a kind of binary notation:

```
sort State
func one: -> State          % the value 1
func x2p0: State -> State   % multiply by 2
func x2p1: State -> State   % multiply by 2 and add 1
```

The function `one` represents the value 1 and the functions `x2p0` and `x2p1` represent the functions 'multiply by 2', respectively 'multiply by 2 and add 1'. One can think of a subterm of sort `State` as a binary number with the most significant bit at the right (which is the reverse of the normal order). For example, `x2p0(x2p1(one)))` corresponds to 110_B (1 for one, 1 for the x2p1 and 0 for x2p0).

II.5. Linearisation of the pCRL part - general method

By default, the lineariser of the tool set applies the method described in [GPU01]. This method transforms the declarations into a 'pre-linear' normal form, the *Extended Greibach Normal Form* (EGNF) before combining them in the manner described above (Section II.4). The single process operator resulting from the combination of several declarations in EGNF form is subsequently linearised.

Hence, the entire default linearisation process consists of the following steps:

1. The pCRL is transformed into one LPO, while the second part is appropriately updated. This step consists of the following sub steps:
   (a) The declarations of the pCRL part are transformed into EGNF.
   (b) The declarations of the pCRL part (now in EGNF) are combined into a single process and the second part is appropriately updated (as described in Section II.4).
   (c) The pCRL part (now consisting of a combined process, the parts of which are in EGNF) is linearised and the parallel pCRL part is appropriately updated.
2. The pCRL part (now consisting of a single LPO) and the parallel pCRL are integrated into one single LPO.

A declaration in EGNF resembles a linear declaration, except that instead of a single recursive process call one can have a sequential composition of process calls. The summands of an LPO all have the form `a.X <| condition |> delta`. The "then" part of declarations in EGNF can also have forms like `a.X.Y.Z`, `a.Y.Z`, and `a.Y.Z.X`. However, forms like `a.b.Y.Z` and `a.Y.b.Z` are not allowed.

The default method uses stacks to linearise the single process equation that results from combining several declarations in EGNF.

As an example consider the following process specification:

```
proc Y = a.Y.b + c
```

This specification describes a process that can do zero or more `a` actions, followed by a `c` action after which as many `b` actions are executed as there were `a` actions.

In order to linearise this process it is first brought into EGNF (step 1a):

```
proc Y = a.Y.Z + c
     Z = b
```

Next, the two declarations are combined into a single declaration (step 1b):

```
X(state: State) = (a.X(Y).X(Z) + c) <| eq(state,Y) |> delta
                +  b                 <| eq(state,Z) |> delta
```

In order to linearise this declaration (step 1c) it is important to note that the state of the process can be described as a sequence of process names. For example, `X(Y)` can perform an `a` action to change into `X(Y).X(Z)`, which can perform another `a` action to change into `X(Y).X(Z).X(Z)` etc. This sequence of process names is represented by means of a sort `Stack`. On this stack entities of sort `State` are pushed. The functions `getstate`, `pop` and `isempty` behave as indicated.[4]

```
sort Stack
func push:State#Stack   ->Stack
     emptystack:        ->Stack
map  getstate:Stack     ->State % the state on the top of the
                                %   stack
     isempty:Stack      ->Bool  % true if and only if the stack
                                %   is empty
     pop:Stack          ->Stack % the stack without the top state
```

Now, the specification can be linearised, with the following result:

```
X(s: Stack) =
        a . X(push(Y,(push(Z,emptystack))))
              <| isempty(s) |> delta
      + c     <| isempty(s) |> delta
      + a . X(push(Y,(push(Z,pop(s)))))
              <| and(eq(getstate(s),Y),not(isempty(s))) |> delta
      + c . X(pop(s))
              <| and(eq(getstate(s),Y),not(isempty(s))) |> delta
      + b . X(pop(s))
              <| and(eq(getstate(s),Z),not(isempty(s))) |> delta
```

In the example above the process to be linearised (`X(state:State)`) does not have parameters other than the `State`. If there are parameters the values of all these parameters must be pushed on the stack. The specification of the data type `Stack` is appropriately modified.

[4]These names are the names used by the tool set.

II.6. Linearisation of the pCRL part - regular methods

In the general method the declarations of the pCRL part of a specification are first brought into EGNF, then combined into a single process description, which is subsequently linearised with help of a stack sort. The use of stacks has a considerable disadvantage: as all data parameters are distributed over different stack frames the LPO is difficult to understand and analyse. However, in many cases, a process specification can be linearised straight away. If the declarations that make up a linearised specification are combined (in the manner described in Section II.4) the result is an LPO. There is no need to introduce stacks. This method of linearisation is called *regular linearisation*.

Regular linearisation consists of the following steps:

1. The pCRL part is transformed into a single LPO, while the parallel pCRL part is appropriately updated. This is done by means of the following sub steps:
   (a) The pCRL part is linearised.
   (b) The (now linear) declarations of the pCRL part are combined into a single LPO, and the parallel pCRL part is appropriately updated.
2. The pCRL part (now consisting of a single LPO) and the parallel pCRL part are integrated into one single LPO.

Regular linearisation results in an LPO that is much easier to analyse (by tools such as `parelm` and `constelm`) than an LPO resulting from a linearisation with stacks. However, in contrast with the default method the regular linearisation is not guaranteed to succeed if the specification meets the requirements.

In the next sub Section (II.6.1) the notion of a linear specification is described. An example illustrates that combination of the declarations of a linear specification results in an LPO (step 1b). Section II.6.2 discusses two methods to linearise a specification (step 1a). Section II.6.3 addresses the question which method is to be used.

II.6.1 Linear specifications

A process specification is *linear* if every process equation has the form `X=a <| condition |> delta` or `x = a.X <| condition |> delta` (the condition is not required).

As discussed in Section II.5 if a specification in EGNF is combined into a single process that process must subsequently be linearised in order to get an LPO. In contrast, a linear specification can be converted into an LPO simply by combining its process declarations in the manner described above (Section II.4).

Consider, for example, the following linear specification:

```
proc A = a . B
     B = b . B + b . A
```

Combining `A` and `B` into `X` gives:

```
proc X(state: State) =   a . X(B) <| eq(state,A) |> delta
                       + b . X(B) <| eq(state,B) |> delta
                       + b . X(A) <| eq(state,B) |> delta
```

This is indeed an LPO.

II.6.2 Linearisation of regular specifications

In basic process algebra (BPA), a process specification is called *regular* if the associated state space is finite. [MM94] describes a method to transform a regular specification into a linear specification.

Consider the following example:

```
proc P = a . B . P
     B = b . B + b
```

This specification describes a process that repeatedly performs an `a` action followed by one or more `b` actions. In order to linearise this specification first replace `B.P` in `P` by `X`:

```
proc P = a . X
     B = b . B + b
     X = B . P
```

In order to guard the declaration of `X`, `B` is expanded. This results in `X = (b.B + b).P` which can be rewritten as `X = b.B.P + b.P` (by axiom A4 of BPA). As `X = B.P`, `B.P` can be replaced by `X` which results in: `X = b.X + b.P`. This gives the following linear specification:

```
proc P = a . X
     B = b . B + b
     X = b . X + b . P
```

If BPA is extended with data (as in μCRL), regular linearisation is more complicated. The lineariser of the tool set provides two methods for regular linearisation, `-regular` and `-regular2`. These methods differ in the way in which the parameters of the newly created process names are generated:

- With the `-regular` method the parameters of the newly introduced process names are the variables of the part to be replaced by that new process.
- With the `-regular2` method a new parameter is generated for each parameter of the part to be replaced.

Consider for example the following process specification:

```
proc R(b: Bit, queue: List) =
       sum(d: D,
             read(b,d) . ack(b) . R(invert(b), add(d,queue)))
          <| not(is-full(queue)) |>
             delta
```

This specification defines a process that reads a frame consisting of a an alternating bit and a datum, acknowledges the receipt of that frame and adds the datum to a bounded queue.

In order to linearise this specification `ack(b).R(invert(b),add(d,queue))` is to be replaced by a process `X`. The flag `-regular` will use the variables `b:Bit`, `d:D`, and `queue:List` as parameters of `X` and invoke `X` as `X(b,d,queue)`:

```
proc R(b: Bit, queue: List) =
       sum(d: D,
             read(b,d) .X(b,d,queue))
          <| not(is-full(queue)) |>
             delta
     X(b: Bit, d: D, queue: List) =
        ack(b) . R(invert(b), add(d,queue))
```

The flag `-regular2` will introduce parameters `b1:Bit`, `b2:Bit`, and `q3:List` for respectively `b`, `invert(b)` and `add(d,queue)` and will invoke `X` as `X(b, invert(b), add(d,queue))`:

```
proc R(b: Bit, queue: List) =
       sum(d: D,
             read(b,d) . X(b, invert(b), add(d,queue)))
```

```
        <| not(is-full(queue)) |>
           delta
   X(b1: Bit, b2: Bit, q3: List) = ack(b1) . R(b2,q3)
```

In μCRL there are many cases in which regular linearisation can be successfully applied to processes that have an infinite state space. For example, if, in the specification above, the bounded queue is replaced by an unbounded one the associated state space is infinite, but it can be linearised with the `-regular` and `-regular2` methods in the same way as in the example above. The same is the case if the alternating bit is replaced by a frame number. The opinions differ on the issue whether or not such processes should be called 'regular'.[5]

II.6.3 Regular versus regular2

The `-regular` and the `-regular2` methods each have advantages and disadvantages:

1. Typically, the use of the `-regular` flag results in a specification with substantially less data parameters than the use of the `-regular2`. This is an advantage because both the lineariser and the instantiator proceed faster.
2. The `-regular2` method is especially useful when a specification contains lots of similar structured process expressions that only differ in the data expressions they contain. Under this conditions linearisation with `-regular2` may result in a specification with a smaller associated state space (smaller than the state space resulting from `-regular`).
3. There are cases in which linearisation with the `-regular2` flag terminates and linearisation with the `-regular` flag not.

I'll discuss each of these points in the given order.

Ad. 1 – the number of parameters Typically, many parameters are used in more than one data expression. As a result `-regular` linearisation will usually generate an LPO with substantially less parameters than `-regular2` linearisation.

There are, however, cases in which `-regular2` linearisation results in an LPO with less parameters than `-regular` linearisation, as is shown by the following example:

```
proc P = sum(d: D, sum(e: D, a . b(f(d,e)) . P))
```

`-regular` will introduce an `X` with two parameters `d` and `e`; `-regular2` will introduce one parameter `v1`.

Ad. 2 – structural similarities In general, if a specification contains several similar structured subterms linearisation with `-regular2` may result in an LPO with a smaller associated state space than an LPO generated by `-regular`.[6] The following specification is a case in point:

```
proc R(b: Bit) =
         sum(d: D,  read(b,d) . deliver(d) . ack(b) . R(invert(b))
                  + read(invert(b),d) . ack(invert(b)) . R(b))
```

This specification defines a process that reads a frame consisting of an alternating bit and a datum. If the expected frame arrives, the datum is delivered to an application, after which the receipt of that frame is acknowledged. If an unexpected (i.e. previous) frame arrives, it is acknowledged and the process continues unaltered.

Linearisation starts by replacing `deliver(d).ack(b).R(invert(b))`. As will be clear from the discussion above, `-regular` does so by means of `X(d:D, b:Bit)`:

[5] Intuitively, regular linearisation is possible if the control of a process is finite (no matter whether or not there are infinite data). Unfortunately, up to know all attempts to formalise this intuition and prove it failed.

[6] We found reductions by 10–15%

```
proc R(b: Bit) =
         sum(d: D,  read(b,d) . X(d,b)
                  + read(invert(b),d) . ack(invert(b)) . R(b))
     X(d: D, b: Bit) =  deliver(d) . ack(b) . R(invert(b))
```

Linearisation with `-regular2` will introduce a process `X(d1:D, b2:Bit, b3:Bit)` for the same purpose:

```
proc R(b: Bit) =
         sum(d: D,  read(b,d) . X(d, b, invert(b))
                  + read(invert(b),d) . ack(invert(b)) . R(b))
      X(d1: D, b2: Bit, b3: Bit) = deliver(d1) . ack(b2) . R(b3)
```

Next, `ack(d).R(invert(b))`, respectively `ack(b2).deliver(d3)`, is to be replaced. The `-regular` flag results in:

```
proc R(b: Bit) =
         sum(d: D,  read(b,d) . X(d,b)
                  + read(invert(b),d) . ack(invert(b)) . R(b))
     X(d: D, b: Bit) = deliver(d) . Y(b)
     Y(b: Bit) =  ack(b) . R(invert(b))
```

The `-regular2` flag gives:

```
proc R(b: Bit) =
         sum(d: D,  read(b,d) . X(d, b, invert(b))
                  + read(invert(b),d) . ack(invert(b)) . R(b))
     X(d1: D, b2: Bit, b3: Bit) = deliver(d1) . Y(b2,b3)
     Y(b1: Bit, b2: Bit) = ack(b1) . R(b2)
```

Next, `ack(invert(b)).R(b)` is to be replaced. Note, that this subterm is similar in structure to the subterm replaced in the previous step. In order to replace this subterm `-regular` has to introduce a new process name `Z(b:Bit)`:

```
proc R(b: Bit) =  sum(d: D,  read(b,d) . X(d,b)
                    + read(invert(b),d) . Z(b))
     X(d: D, b: Bit) = deliver(d) . Y(b)
     Y(b: Bit) = ack(b) . R(invert(b))
     Z(b: Bit) = ack(invert(b)) . R(b)
```

The `-regular2` method needs a `Z(b1:Bit, b2:Bit) = ack(b1).R(b2)`. There is already such a process declared, namely `Y`, and `-regular2` will use that:

```
proc R(b: Bit) =
         sum(d: D,   read(b,d) . X(d, b, invert(b))
                   + read(invert(b),d) . Y(invert(b), b))
     X(d1: D, b2: Bit, b3: Bit) = deliver(d1) . Y(b2,b3)
     Y(b1: Bit, b2: Bit) = ack(b1) . R(b2)
```

It is easy to see that the state space associated with the specification generated with `-regular2` has a smaller state space than the one associated with `-regular` (fig. II.3). Assume that there is only one datum (`d`). Starting with R(0), a `read(0,d)` can be executed resulting in `X(0)` (`-regular`) respectively `X(0,1)` (`-regular2`). From this state a `deliver(d)`, can be executed (resulting in `Y(0)`, respectively `Y(0,1)`), followed by a `ack(0)` resulting in `R(1)` in both cases. At this point, if a `read(0,d)` is executed `-regular` will enter state `Z(0)`, but `-regular2` will return to `Y(0,1)`. Similar considerations apply to what happens if `read(1,0)` is executed from state `R(0)`.

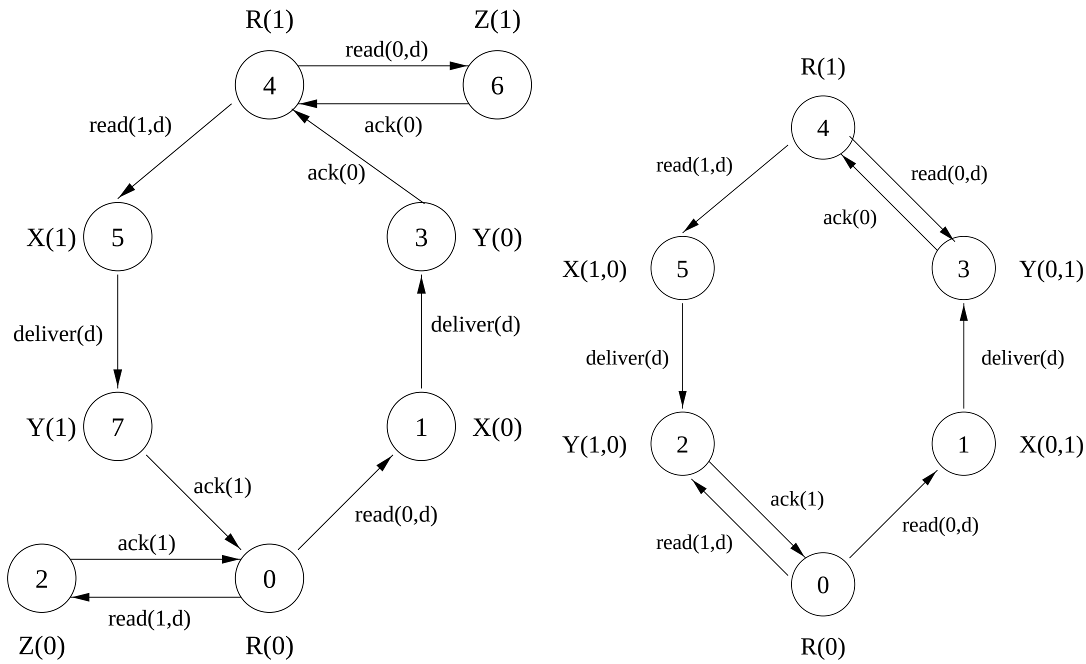

Figure II.3: The state spaces with `-regular` (left) and `-regular2` (right)

Ad. 3 – termination There are cases in which linearisation with the `-regular2` method terminates and linearisation with the `-regular` flag not. Here is an example:

```
proc P(n: Nat, m: Nat) = a . B(succ(n)) . P(n,succ(m))
     B(n: Nat) = b . B(succ(n)) + c
```

Linearisation with the `-regular` flag will start by introducing an `X(n:Nat, m:Nat)` to replace `B(succ(n)).P(n,succ(m))`:

```
proc P(n: Nat, m: Nat) = a . X(n,m)
     B(n: Nat) = b . B(succ(n)) + c
     X(n: Nat, m: Nat) = B(succ(n)) . P(n, succ(m))
```

Next `B(succ(n))` of `X` must be expanded in order to guard the declaration of `X`. After application of axiom A4 this gives:

```
proc P(n: Nat, m: Nat) = a . X(n,m)
     B(n: Nat) = b . B(succ(n)) + c
     X(n: Nat, m: Nat) = b . B(succ(succ(n))) . P(n, succ(m))
                       + c . P(n, succ(m))
```

In the next step `B(succ(succ(n))).P(n,succ(m))` will have to be replaced. As there is no suitable `X`, a new process name, say `Y(n:Nat, m:Nat)`, is introduced, which must be guarded subsequently, and so on ad infinitum.

However, linearisation with `-regular2` will terminate. First `B(succ(n)). P(n,succ(m))` will be replaced by `X(n1:Nat, n2:Nat, n3:Nat)`:

```
proc P(n: Nat, m: Nat) = a . X(succ(n), n, succ(m))
     B(n: Nat) = b . B(succ(n)) + c
     X(n1: Nat, n2: Nat, n3: Nat) = B(n1) . P(n2,n3)
```

Then `X` is guarded by expanding B(n1). After application of axiom A4 this gives:

```
proc P(n: Nat, m: Nat) = a . X(succ(n), n, succ(m))
     B(n: Nat) = b . B(succ(n)) + c
     X(n1: Nat, n2: Nat, n3: Nat) = b . B(succ(n1)) . P(n2,n3)
                                  + c . P(n2,n3)
```

Note that `B(succ(n1)).P(n2,n3)` = `X(succ(n1), n2, n3)`. The application of this equality to replace `B(succ(n1)).P(n2,n3)` in the declaration of `X` results in the following process specification:

```
proc P(n: Nat, m: Nat) = a . X(succ(n), n, succ(m))
     B(n: Nat) = b . B(succ(n)) + c
     X(n1: Nat, n2: Nat, n3: Nat) = b . X(succ(n1), n2, n3)
                                  + c . P(n2,n3)
```

This specification is linear.

In theory, the situation that linearisation with `-regular` does not terminate whereas linearisation with `-regular2` does, occurs only if the associated state space is infinite. If the state space is finite there will be some point in the replacement cycle where a suitable process name already exists. However, as the specification is not rewritten during linearisation the lineariser may not recognise this. This means that in practice it is always worth to try whether `-regular2` terminates if `-regular` does not. For example, if in the example above the sort `Nat` is replaced by a data type `Mod-2` defined in the following way:

```
sort Mod-2
func 0,1: -> Mod-2
map  succ: Mod-2 -> Mod-2
rew  succ(0) = 1
     succ(1) = 0
```

the associated state space is finite but `-regular` will not terminate as the lineariser will not recognise that `succ(succ(0))` is the same as `0`. Similarly, if the function `succ` is replaced by some function `succ-mod2:Nat->Nat` which computes n mod 2 the associated state space is finite but linearisation with the `-regular` flag will not terminate as the lineariser does not recognise that e.g. `succ-mod2(succ-mod2(succ-mod2(0)))` equals to `0`.

II.7. Parallel composition

After the pCRL part is transformed into an LPO and the parallel pCRL part is properly updated the two parts are integrated into a single LPO. This transformation is usually called *parallel composition.*

One of the advantages of LPOs is that parallel composition of two LPOs comes down to the concatenation of the list of summands of each process. To see this, suppose `Z(S)` is an LPO and `Z` and `P = Z(X) || Z(Y)` are to be integrated into a single LPO. In order to do so a new process name `U(S1,S2)=Z(S1)||Z(S2)` is introduced. Now `P` can be written as `U(X,Y)`. By axiom CM1

```
U(X,Y) =   Z(X) ||_ Z(Y)
         + Z(Y) ||_ Z(X)
         + Z(X) | Z(Y)}
```

Consider the first summand (`Z(X) ||_ Z(Y)`). As `Z` is linear `Z(X)` consists of summands the "then-par" of which have the form `a.Z(K)` or `a`. According to axiom CM4 each of these summands can be left merged separately to `Z(Y)`. So, the first summand (`Z(X) ||_ Z(Y)`) can be replaced by a series of summands each has either the form `a.Z(K)||_Z(Y)` or `a||_Z(Y)`. According to axiom CM3 summands of the first type equal to `a.(Z(K)||_Z(Y))` which can be rewritten as `a.U(K,Y)`. Summands of the second type equal to `a.Z(Y)`. Similar considerations apply to the second (`(Y) ||_ Z(X)`) and the third summand (`Z(X) | Z(Y)`). The result at this point is a series of summands each consisting of an action followed by `U` or `Z`. In order to transform this into an LPO, `U` and `Z` must be combined into a new process, say `V`, in the manner described in Section II.4.

Appendix III
Rewriting

The equations specified with the keyword **rew** are applied by `msim`, `instantiator` and `rewr` to rewrite (reduce) the specification. These tools use the equations to generate c-code for reducing data terms. This code is compiled using the gnu c-compiler (`gcc`) and dynamically linked to the relevant tool.

This chapter describes how the equations are applied (Section III.1), and discusses some problems related to the rewriting method (Section III.2) and to the way in which the rewriter is used by the `instantiator` (Section III.3).

III.1. Method

- Currently, the rewriter applies the equations of **rew** sections as rewrite rules from left to right.
- The rewriter attempts to rewrite the arguments of a function before it rewrites the function as a whole (this is called *innermost* rewriting).
- If more than one argument can be rewritten by application of the rewrite rules the rewriter will choose the *leftmost* argument.
- If more than one rewrite rule applies to the same term the rewriter will use the one that appears first in the specification.

This way of using the equations is not prescribed in the definition of μCRL and it may change in the future. More than that, a change is planned for the last two usage rules. It is therefore highly recommended that a specification is written in such a way that the results of the application of the rewrite rules depend neither on the order in which the terms appear nor on the order in which the rules appear.

III.2. Innermost rewriting

Some algebraic equations do not terminate when applied as rewrite rules in innermost rewriting. For example, the modulo function `mod` can be specified algebraically as

```
mod(m,n) = if (ge(m,n),mod(sub(m,n),n),m)
```

However, if this equation is used as a rewrite rule, innermost rewriting will not terminate. To understand this, assume the following specification:

```
sort Nat
func 0: -> Nat
     s: Nat -> Nat

map ge: Nat # Nat -> Bool  % greater than or equal to
var n,m: Nat
rew ge(n,0) = T                                   % ge-1
    ge(0,s(0)) = F                                % ge-2
    ge(s(n),s(m)) = ge(n,m)                       % ge-3

map if: Bool # Nat # Nat  % if(Bool,m,n) is m if Bool is T
                          %                 n if Bool is F
var n,m: Nat
rew if(T,m,n) = m                                 % if-1
    if(F,m,n) = n                                 % if-2

map sub: Nat # Nat -> Nat     % sub(m,n) means m minus n
var n,m: Nat
rew sub(n,0) = 0                                 % sub-1
    sub(s(n),s(m)) = sub(n,m)                    % sub-2

map mod: Nat # Nat -> Nat     % mod(m,n) means m mod n
rew mod(m,n) = if(ge(m,n),mod(sub(m,n),n),m)  % mod-1
```

and consider what happens if `mod(s(s(s(0))),s(s(0)))` (3 `mod` 2) is rewritten (the sub term that will be rewritten in the next step is underlined):

1. The rewriter will start by applying mod-1 to rewrite

   <u>`mod(s(s(s(0))),s(s(0)))`</u>

   as a whole as:

   `if(`<u>`ge(s(s(s(0))), s(s(0)))`</u>`, mod(sub(s(s(s(0))),s(s(0))),s(s(0))), s(s(s(0))))`

2. The leftmost argument of this function can be rewritten to `T` (applying ge-3 and ge-1). This results in the following term:

   `if(T, mod(`<u>`sub(s(s(s(0))),s(s(0)))`</u>`,s(s(0))), s(s(s(0))))`

3. Algebraically, this whole term can be replaced by `mod(sub(s(s(s(0))), s(s(0)))` (equation if-1), but as the rewriter works innermost, it will not do this. Instead, it will try to rewrite the second argument (the function `mod(sub(s(s(s(0))),s(s(0))),s(s(0)))`). Rule mod-1 applies to this term, but as one of its sub terms (`sub(s(s(s(0))),s(s(0)))`) can also by rewritten (applying sub-1 and sub-2) the innermost rewriter will do that. The result is:

   `if(T, `<u>`mod(s(0),s(s(0)))`</u>`, s(s(s(0))))`

4. The second argument can be rewritten once more (applying mod-1). The result is:

   `if(T, if(`<u>`ge(s(0),s(s(0)))`</u>`,mod(sub(s(0),s(s(0)))),s(s(0))), s(s(s(0))))`

5. The leftmost sub term of the second argument can be rewritten to `F` (using ge-3 and ge-2). The result is:

   `if(T, if(F,mod(`<u>`sub(s(0),s(s(0)))`</u>`,s(s(0)))), s(s(s(0))))`

6. Algebraically `if(F,sub(s(0),s(s(0))),s(s(0)))` could be replaced by `s(s(0))` (if-2), but as rewriting is innermost the rewriter will now apply sub-2 to replace `sub(s(0),s(s(0))` by `sub(0,s(s(0)))`. There are no rules to reduce this further. The result is:

```
if(T, if(F,mod(sub(0,s(s(0))),s(s(0))),s(s(0))), s(s(s(0))))
```

7. The underlined function can be rewritten again using mod-1:

```
if(T, if(F, if(ge(sub(0,s(s(0))), s(s(0))), mod(sub(sub(0,s(s(0))), s(s(0))), s(s(0))),
s(s(0))), s(s(s(0))))
```

8. Rule mod-1 can be applied to the result (note that `ge(sub(0,s(s(0))), s(s(0)))` can not be rewritten):

```
if(T, if(F, if(ge(sub(0,s(s(0))),s(s(0))), if(ge(sub(sub(0,s(s(0))), s(s(0)))),
mod(sub(sub(sub(0,s(s(0))), s(s(0))))),s(s(0))),s(s(0))), s(s(s(0))))
```

9. and so on.

The function `mod` is replaced by a term that contains the function `mod` as one of its sub terms, this sub term in turn is replaced by a term that contains `mod` as one of its sub terms, and so on. Note that, mod-1 does not put any limitation on the form of the arguments of `mod`, every (well-typed) `mod` can be rewritten. As a work around one might limit the possibility to rewrite `mod`, preferably, in such way that it can not be rewritten after the point were the answer is known (step 6). This can be done by replacing mod-1 with the following rules:

```
rew mod(0,n)    = 0                          % mod-2a
    mod(s(m),n) = if(ge(s(m),n),mod(sub(s(m),n),n),s(m))
                                             % mod-2b
```

Now, `mod` can only be rewritten if the first argument is `0` or `s(m)`. This precludes the end-less rewriting of `mod(sub(n),m)`.

Rewriting of `mod(s(s(s(0))),s(s(0)))` will proceed in the same way as above, up to and including 6 (mod-2b is applied instead of mod-1). This results in:

```
if(T, if(F,mod(sub(0,s(s(0))),s(s(0))),s(s(0))), s(s(s(0))))
```

But then a difference occurs. This time there is no rule to replace `mod(sub(0, s(s(0))), s(s(0)))`. Instead, the rewriter will apply if-2 on the underlined function. This results in:

```
if(T, s(s(0))), s(s(s(0))))
```

To this term the rewriter will apply if-1:

```
s(s(0))
```

which is the desired result.

III.3. Rewriting of open terms

Although a mapping is completely specified by equations telling how the mapping applies to the constructors, it it is often useful or even necessary to add rules for boolean open terms. An example of such a rule is:

```
var b: Bool
rew and(b,not(b)) = F
```

In order to understand why such an addition might be useful one must delve into the inner workings of the instantiator.

As said, in Section 3.3 the instantiator generates the state space reachable from the initial state described in the specification. One main component of the instantiator is the stepper. The stepper accepts an input state and a linearised specification and returns a list of (action,state) pairs, representing steps that are possible from the input state. In order to determine which steps are possible the stepper evaluates the condition of each summand. If a summand contains a sum operator the stepper will pass the condition (an open terms of type boolean) to another component, the enumerator. This component enumerates the data terms for which the condition is true.

For example, suppose that a summand contains a sum operator with sum variable `n` of type `Natural` and a condition `le(n,s(s(0)))` ($n < 2$). The stepper will pass the open term `le(n,s(s(0)))` to the enumerator (together with the specification of the relevant data types). The enumerator will try to evaluate this term and if it does not succeed it will try to evaluate all terms in which the variable `n` is replaced by a constructor of its type (`Natural`) applied to a fresh variable. Suppose `le(n,m)` is specified as follows:

```
map le: Nat#Nat -> Nat
var n,m: Nat
rew le(0,0)       = F
    le(0,s(n))    = T
    le(s(n),0)    = F
    le(s(n),s(m)) = le(n,m)
```

Let's see what happens:

- The enumerator will first try to evaluate `le(n,s(s(0)))`.
- As there is no rule for this open term it will not succeed.
- Next, the enumerator will try to evaluate `le(0,s(s(0)))` and `le(s(n'), s(s(0)))` (which are all terms resulting from replacing `n` by one the constructors of `Natural` (`0` and `s(n)`) applied to the fresh variable `n'`).
- The evaluation of `le(0,s(s(0)))` will result in `T` and the enumerator will return `0` as a data term that satisfies the condition
- The second term (`le(s(n'),s(s(0)))`) can be rewritten as `le(n',s(0))`, but no further reduction is possible. Hence the attempt to evaluate `le(s(n'),s(s(0))` does not succeed.
- Therefore, the enumerator will now try `le(s(0),s(s(0)))` and `le(s(s(n'')),s(s(0)))`.
- The first term (`le(s(s(0)),s(s(0)))`) can be rewritten to `T` and is returned to the stepper.
- The attempt to evaluate the second term (`le(s(s(n'')),s(s(0)))`) does not succeed as this term reduces to `le(n'',0)`.
- Therefore, the enumerator will now try `le(s(s(0)),s(s(0))` and `le(s(s(s(n'''))),s(s(0,0))`.
- The first term evaluates to `F`.
- The second term can be reduced to `le(s(n'''), s(s(0)))` which evaluates to `F` as well.
- Hence, the enumerator is done.

It can occur that certain conditionals have a value but that it cannot be evaluated by the enumerator. As an example consider the condition `and(even(n),not(even(n)))`. Obviously, this condition is false. However, in absence of the equation `and(x, not(x)) = F` the enumerator will not be able to evaluate it and will return the error message `Condition %s does not evaluate to T or F`.

Assume the following specification of `even` and `odd`

```
map even, odd: Nat -> Bool
var n: Nat
rew odd(s(0))      = T
    even(s(0))     = F
    odd(s(s(n)))   = even(s(n))
    even(s(s(n))) = odd(s(n))
```

The enumerator will first try to evaluate `and(even(n),not(even(n)))`. It will not succeed. It will then try the terms `and(even(0),not(even(0)))` and `and(even(s(n')),not even(s(n')))`. As, there are no rules for these terms, neither will succeed. The enumerator will now try the terms `and(even(s(0)), not(even(s(0))))` and `and(even(s(s(n''))), not(even(s(s(n'')))))`. The first term reduces to F. The second term will be reduced to `and(odd(s(n'')), not(odd(s(n''))))` which cannot be evaluated. For that reason, the enumerator will try `and(even(s(s(0))),not(even(s(s(0)))))` (which reduces to F) and `and(even(s(s(s(0)))),not(even(s(s(s(0))))))` (which does not succeed). And so on . . . . After a certain number of trials the enumerator will give up as a result of which `instantiator` will exit with the error `Condition and(even(zero),not(even(zero))) does not evaluate to T or F`.

However, if the equation `and(x,not(x)) = F` is added to the specification, the first attempt will succeed, as `and(even(n),not(even(n)))` will be reduced to `F`. The enumerator will inform the stepper that there are no substitutions that satisfy the condition and all goes well.

Here are some examples of other equations that are worth adding:

- `if(b,x,x)=x` (`b` is of sort `Bool`)
- `not(not(b)) = b`

Appendix IV
License

The software of the μCRL tool set may be used provided the following license agreement is obeyed.

IV.1. Preamble

The μCRL tool set version 1.xx (where xx stands for any one or two digit number) is a set of programs to transform and analyze μCRL specifications. This license is intended to make clear that the software may be used freely, both for commercial and non commercial purposes, provided that when it is used in part or as a whole it must be made clear that

1. the μCRL tool set version 1.xx, developed at the theme SEN2 of the Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, Amsterdam, The Netherlands, developed by Jan Friso Groote and Bert Lisser

2. the aterm library version 0.xx, developed at the faculty of mathematics, computer science, physical science and of the University of Amsterdam, Kruislaan 409, Amsterdam, The Netherlands developed by Hayco de Jong and Pieter Olivier.

have been used to deliver certain product or service.

Furthermore, this license excludes any liability caused by direct or indirect use of the software, and any obligation to provide support or updates in any form.

IV.2. License Agreement

Hereinafter "LICENSEE" refers to the person or institution who has made a copy of the file mcrl-1.xx.tar.gz which contains analysis and transformation programs and documentation and a copy of this License Agreement.

Hereinafter "LICENSOR" refers to

The Stichting Mathematisch Centrum, existing under the laws of The Netherlands, and having offices at the

Kruislaan 413 Amsterdam The Netherlands

WHEREAS

LICENSOR has developed a body of computer programs and associated documentation, referred to as "the μCRL tool set version 1.xx" by the authors, and hereinafter referred to as the SOFTWARE,

LICENSOR has made available the SOFTWARE in a file mcrl-1.xx.tar.gz which contains the SOFTWARE and a copy of this Academic License Agreement,

LICENSEE desires to use the SOFTWARE for any purpose,

LICENSEE agrees as follows:

The LICENSOR grants to the LICENSEE a non-exclusive, non-transferable, license to use the SOFTWARE subject to the following conditions.

1. LICENSEE'S RIGHTS. LICENSEE shall have the right to use the SOFTWARE for any purpose. In particular LICENSEE may use the software for the commercial or non-commercial analysis or transformation of μCRL specifications, the construction of commercial or non commercial software based on (parts of) the SOFTWARE, or the construction of commercial or non commercial software analysis packages.

2. RESTRICTIONS ON USE. LICENSEE takes care that whenever the SOFTWARE is used, either directly or indirectly, as a whole or in part for delivering services of any kind, construction of new software or software analysis packages it is made clear that the service or software makes use of:

   (a) the μCRL tool set version 1.xx, developed at the theme SEN2 of the Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, Amsterdam, The Netherlands, developed by Jan Friso Groote and Bert Lisser

   (b) the aterm library version 0.xx, developed at the faculty of mathematics, computer science, physical science and astrophysics of the University of Amsterdam, Kruislaan 409, Amsterdam, The Netherlands developed by Hayco de Jong and Pieter Olivier.

   (This can for instance be done by mentioning 1. and 2. above in a README file, an "about" section, or by referring to the software in an accompanying paper. For services offered, it suffices to make clear either verbally or in writing that above mentioned software has been used)

3. LICENSING FEE. There is no fee for this license.

4. NO SUPPORT. The LICENSEE recognizes that the LICENSOR is not obligated to provide support, maintenance, consulting, or revision of the SOFTWARE. If the LICENSOR chooses to release to the LICENSEE updates of, additions to, or modifications of the SOFTWARE, this agreement shall apply to them as though they were part of the original SOFTWARE.

5. NO PRODUCT WARRANTY. The SOFTWARE is released on an "as is" basis. There is no warranty whatsoever as to functioning, performance or effect on hardware or other software, express or implied. By way of an example, but not limitation, the LICENSOR makes no representations or warranties of merchantability or fitness for any particular purpose or that the use of the licensed SOFTWARE will not infringe any patents, copyrights, trademarks or other rights.

6. OWNERSHIP. The LICENSEE agrees that the SOFTWARE, including any updates, additions, and modifications, is, and shall at all times remain, the property of the LICENSOR, and that it has been copyrighted by the LICENSOR. The LICENSEE shall have no right, title or interest therein or thereto except as expressly set forth in this agreement.

7. CREDITS. All credits and copyright notices in the SOFTWARE, both in listings and/or documentation, whether names of individuals or organizations, shall be retained in place. Publications referring to the SOFTWARE, or to other SOFTWAREs containing the SOFTWARE in whole or in part, shall refer to it as the μCRL tool set version 1.xx and the aterm library version 0.xx, and shall specify that the SOFTWARE was obtained under license from the Stichting Mathematisch Centrum in Amsterdam.

8. NO LIABILITY. Neither the LICENSOR nor any individual or any legal entity involved in creating, modifying, updating, or supplementing the SOFTWARE, shall be liable for damages arising out of the failure or malfunctioning of the SOFTWARE. The LICENSEE hereby assumes the risk of and releases and forever discharges the LICENSOR, their employees and any other individual or legal entity referred to in the foregoing sentence with respect to any expense, claim, liability, loss or damage, direct or indirect, including any incidental or consequential damages, whether made or suffered by LICENSEE or by third parties obtaining access to the SOFTWARE through LICENSEE in connection with the failure or malfunction of the SOFTWARE. LICENSEE acknowledges that the SOFTWARE is in the process of development and is not error-free, that the foregoing exclusion of liability is therefore an essential term of this Agreement without which exclusion the LICENSOR would not be willing to enter into this Agreement and to make the SOFTWARE available for free.

9. ACCEPTANCE. Installing or using of any part of the SOFTWARE implies acceptance of the terms and conditions of this agreement.