Computer Assisted Manipulation of Algebraic Process Specifications

J.F. Groote, B. Lisser

Software Engineering (SEN)

SEN-R0117 May 31, 2001

Report SEN-R0117
ISSN 1386-369X

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

Computer Assisted Manipulation of Algebraic Process Specifications

Jan Friso Groote and Bert Lisser
J.F.Groote@tue.nl, Bert.Lisser@cwi.nl

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

ABSTRACT

Specifications of system behaviour tend to become large. Analysis of such specifications requires automated tools. Most attention hitherto has been invested in fully automatic tools. We however believe that in many cases human intervention is required and we therefore propose a number of computer tools to transform process specifications. The concrete manipulation tools that we describe can eliminate constants, redundant sum variables and parameters, and allow to split variables ranging over complex datatypes. These tools can transform specifications with large finite state spaces to variants with state spaces being a fraction of their original size, and transform specifications with infinite state spaces to those with finite state spaces.

2000 Mathematics Subject Classification: 68M14, 68Q60, 68Q85
Keywords and Phrases: Automated Reasoning, Distributed systems, Linear Process Equations, Model Checking, Verification
Note: Research carried out in SEN2, with financial support of the "Systems Validation Center".

1. Introduction

Tools and techniques for the analysis of system behaviour become increasingly powerful. Currently, we can regularly, automatically and thus efficiently answer questions about systems with a restricted state space, and we can also occasionally obtain insight in the behaviour of more substantial systems. But the results in this last category, often require ingenuity, insight in the problem domain, and often the ad hoc design or adaption of tools.

We believe that this is typical for the study of more complex systems. In the first place, fully automatic analysis tools are generally not able to perform the analysis of complex systems fully on their own. This explains why human ingenuity is required in the process. In the second place, it is out of the question that these systems be analysed by hand only. The size of such descriptions may literally stretch to up to thousands of pages. This is why automation is needed.

We think that it is not desired that for each complex system to be analysed in the future, new transformation programs must be designed, or old analysis programs must be adapted. It is better to provide a set of manipulation tools that can be used to preprocess and transform a given system such that it fits a standard analysis tool.

Note that such a situation is not very uncommon in the field of mathematical analysis. Typical mathematical analysis tools such as *Mathematica* and *Maple* do not attempt to solve a formula on their own, but they operate under strict supervision of a user, who, when sufficiently skilled in the use of such a tool, can lead the system to an answer.

In this paper we describe a number of transformation tools that we have developed. We show how in certain cases we can manipulate descriptions of distributed systems to reduce their state spaces from infinite to finite, or to a fraction of their original size.

We work in the setting of process algebra with data μCRL [10], or more precisely in the setting of linear process equations. The language μCRL is a straightforward process algebra, based on ACP_τ [2],

which means that it comprises operators such as $+, \cdot, \|, \tau_I$ and ∂_H extended with equational abstract datatypes. In order to combine data with processes the then_if_else operator $\_ \lhd \_ \rhd \_$ and the choice over a possibly infinite datatype $\sum_{d:D}$ have been added, and parametric process variables and actions are allowed. This language is rather compact, but sufficiently complete to describe virtually any distributed system. When investigating systems described in μCRL our current standard approach is to transform these processes to linear process equations (LPEs). In essence this is a vector of data parameters and a list of condition, action and effect triples that describes when an action may happen, and what its effect is on the vector of data parameters. An LPE is a special instance of a μCRL process. In [11, 15] it is described how a large class of μCRL processes can be transformed automatically to LPE format in such a way that the resulting LPE is strongly bisimilar to the original. Our transformation tools work strictly on LPEs.

There are four tools that we describe here:

Elimination of constant process parameters (Subsection 4.1)**.** A parameter of an LPE can be replaced by a constant value, if that parameter remains constant throughout any run of the process.

Elimination of sum variables (Subsection 4.3)**.** It happens that the choice over infinite datatypes is restricted in the condition to a single concrete value. In that case it is more efficient to replace the sum variable by this single value.

Elimination of dead process parameters (Subsection 4.2)**.** A parameter of an LPE can be removed, if that parameter does not influence the behaviour of a process (has neither directly or nor indirectly influence on actions and conditions). We call such a parameter a dead parameter. Whereas the first two reduction techniques only simplify the description of an LPE, this one also allows substantial reduction of the state space underlying an LPE. Actually, if the dead process parameter ranges over an infinite domain, the state space can be changed from infinite to finite by this operation.

Elimination of structured variables (Subsection 4.5)**.** It is not always possible to apply the operations mentioned above to single variables, but we can apply them to parts of the data structures that these variables range over. For this to take place, we must partition such data structures into its constituents.

The reduction methods have been implemented in the μCRL toolset [17]. We explain all the reduction techniques and show that they are sound in the sense that they maintain strong bisimulation. Furthermore, in some cases we show the strength of the algorithm, by providing a theorem to which extent the transformation can be considered complete. We show how the tools interact restricting the sequences in which they need to be applied and we show the effect of the tools on a number of rather diverse communication protocols.

Currently, we are working on a next generation of these tools. The reasoning engine we use is an extremely fast rewriting engine built according to the ideas in [1]. We found that with an adequate theorem prover [14] , we can on the one hand increase the power of the current tools, and on the other hand build additional tools, for instance those based on confluence reduction [5] [12] of which first prototypes are very promising.

2. Preliminaries

The *Micro Common Representation Language* [10], *μCRL*, has been defined to describe interacting processes that rely on data. This language includes a formalism for algebraic specification of abstract data types (data part) and the Algebra of Communicating Processes with abstraction, *ACP* [6] [2] (process part). An algebraic specification consists of a signature which contains the definitions of

abstract data types, constructors, and operators, and a set of equations. In subsections 2.1 and 2.2 we describe the data part. In the subsequent subsections we describe linear process equations and bisimulation.

2.1 Algebraic specifications of abstract data types

In this subsection we describe the basic aspects of abstract data types in a standard way, see e.g. [16]. We make a distinction between 'ordinary' functions, which we call *mappings*, and *constructors*, with the requirement that all terms of a sort that contains constructors can be written using constructors only.

Throughout the text we assume the existence of an infinite set V of variable declarations of the form x: $\mathbf{S}$, where $\mathbf{S}$ is some sort.

Definition 2.1. A *signature* is a triple $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{M})$ where

- $\mathcal{S}$ is a set of *sorts*. We assume that **Bool** is an existing sort, i.e. $\mathbf{Bool} \in \mathcal{S}$;
- $\mathcal{F}$ is a set of *constructors*, i.e. functions of the form $f\colon \mathbf{S}_1 \times \cdots \times \mathbf{S}_n \to \mathbf{S}$ and $\mathbf{S}_i, \mathbf{S} \in \mathcal{S}$. Moreover, we presuppose the existence of the constructors $\mathsf{t}, \mathsf{f} :\to \mathbf{Bool}$ in $\mathcal{F}$; t stands for "true" and f for "false".
- $\mathcal{M}$ is a set of *mappings*, i.e. functions of the form $f\colon \mathbf{S}_1 \times \cdots \times \mathbf{S}_n \to \mathbf{S}$ and $\mathbf{S}_i, \mathbf{S} \in \mathcal{S}$. The sets $\mathcal{F}$, $\mathcal{M}$ and V are pairwise disjoint.

We call $\mathbf{S} \in \mathcal{S}$ a Σ-*constructor sort* iff there is a function symbol $f\colon \mathbf{S}_1 \times \cdots \times \mathbf{S}_n \to \mathbf{S}$ in $\mathcal{F}$. A Σ-*term* is a term defined over the signature Σ in the standard way.

We use substitutions mapping variables to terms with the same sort. We typically use either the letter ρ, or $[x{:=}t]$ for substitutions.

Definition 2.2. An *abstract data type* is a tuple $\mathcal{A} = (\Sigma, \mathcal{E})$ where

- $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{M})$ is a signature;
- $\mathcal{E}$ is a set of *equations* (generally used as *rewrite rules*), i.e. expressions of the form $t = u$ with both t and u Σ-terms of some sort $\mathbf{S} \in \mathcal{S}$.

We specify data types using the syntax of μCRL [10]. The function symbols that follow the keyword **func** are constructors. The function symbols that follow **map** are mappings. Here follows an example of an algebraic specification of the datatype **Bool**.

Example 2.3. The data type associated with the sort **Bool** consists of the constants t (true) and f (false). Mappings $\neg$ and $\wedge$ are specified.

sort **Bool**
func $\mathsf{t}, \mathsf{f} :\to \mathbf{Bool}$
map $\neg : \mathbf{Bool} \to \mathbf{Bool}$, $\wedge : \mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool}$
var $b : \mathbf{Bool}$
rew $\neg\mathsf{t} = \mathsf{f}$, $\neg\mathsf{f} = \mathsf{t}$, $\mathsf{t} \wedge b = b$, $\mathsf{f} \wedge b = \mathsf{f}$

2.2 Interpretation of abstract data types

We interpret an abstract data type using model class semantics, of which we describe only the main ingredients here (see [8] for a detailed account).

Let $\mathcal{A} = (\Sigma, \mathcal{E})$ be an abstract data type with $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{M})$ a signature. For each $\mathbf{S} \in \mathcal{S}$ there is a non empty domain $\mathbf{D_S}$ and a valuation σ that maps variables of sort $\mathbf{S}$ to elements of $\mathbf{D_S}$. $\mathbb{M} = \{\mathbf{D_S} | \mathbf{S} \in \mathcal{S}\}$ is a model of $\mathcal{A}$ iff for every sort $\mathbf{S}$ of the abstract data type there is some interpretation $[\![\cdot]\!]^\sigma$ that maps each term of sort $\mathbf{S}$ to an element of $\mathbf{D_S}$, coincides with σ on the variables, and respects the equations $\mathcal{E}$. An interpretation $[\![\cdot]\!]^\sigma$ respects the equations $\mathcal{E}$ iff the following applies: if $t = u \in \mathcal{E}$ then $[\![t]\!]^\sigma = [\![u]\!]^\sigma$.

If $\mathbf{S}$ is a constructor sort, then each element of $\mathbf{D_S}$ must be equal to a term consisting of constructors only, possibly applied to elements of non constructor domains. We say two terms are equal, notation $t = u$ when for every model of $\mathcal{A}$ and interpretation function t and u are interpreted as the same element in the domain. We can reason about equality using standard equational logic enhanced with induction principles for constructors.

For the sort **Bool** with elements `t` and `f` we assume that $\mathbf{D_{Bool}}$ is a domain with exactly two elements representing `t` and `f`. We also assume that for each sort $\mathbf{S}$ there is a mapping $\doteq\colon \mathbf{S} \times \mathbf{S} \to \mathbf{Bool}$ that represents equality between terms of sort $\mathbf{S}$. I.e., $t \doteq u = \mathtt{t}$ iff $t = u$.

2.3 Linear Process Equations

We specify processes in μCRL, which is ACP [6] [2], comprising in essence actions and the operators $+$, $\cdot$, $\|$, ∂_H, τ_I, δ, extended with abstract data types. Furthermore, actions and processes can be parameterised with data, and there are two new operators, namely $\sum_{d:D}$, the sum over possibly infinite data types, and $\_ \lhd \rhd \_$ which is the then-if-else operator. Despite the fact that μCRL is a straightforward language, we want an even more straightforward form to facilitate analysis. This basic form is called a linear process equation (LPE) and it consists essentially of a state vector of typed variables and a list of condition-action-effect triples. The LPE basic format is particularly powerful, because it allows parallel processes to be combined without the *state space explosion* effect that occurs in automata. There are effective algorithms to transform process specifications to LPEs [11]. Recently specifications with 250 parallel components and over thousand pages of description have been transformed to LPE format [4].

We want an LPE to be a μCRL process itself, explaining the process algebraic formulation of the condition-action-effect rules below:

Definition 2.4. Let $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{M})$ be the signature defined in the data part. A *linear process equation (LPE)* over Σ is an expression of the form:

$$X(d_1{:}D_1, \ldots, d_n{:}D_n) = \sum_{i \in I} \sum_{e_1^i : E_1^i} \cdots \sum_{e_{n_i}^i : E_{n_i}^i} a_i(f_i)\, X(g_1^i, \ldots, g_n^i) \lhd c_i \rhd \delta$$

where I is a finite index set, and for each $i \in I$:

- $d_1, \ldots, d_n, e_1^i, \ldots, e_{n_i}^i$ are pairwise different variables;
- $D_1, \ldots, D_n, E_1^i, \ldots, E_{n_i}^i \in \mathcal{S}$;
- f_i is a Σ-term in which variables $d_1, \ldots, d_n$ and $e_1^i, \ldots, e_{n_i}^i$ may occur;
- for $1 \leq j \leq n$, g_j^i is a Σ-term of sort D_j in which variables $d_1, \ldots, d_n$ and $e_1^i, \ldots, e_{n_i}^i$ may occur;
- c_i is a Σ-term of sort **Bool** in which variables $d_1, \ldots, d_n$ and $e_1^i, \ldots, e_{n_i}^i$ may occur.

Sometimes we provide an initial state $s_1, \ldots, s_n$ for an LPE. It is convenient to use the vector notation $\vec{s}$ for $s_1, \ldots, s_n$. Consequently, we speak about the initial vector. The process $X(\vec{s})$ represents the behaviour of X from initial state $\vec{s}$. In μCRL a process definition, and hence an LPE, is preceded with the keyword `proc` and an initial state with the keyword `init`. See e.g. the example in Section 4.4.

2.4 Bisimulation

All the reduction methods that we present preserve strong bisimulation. Therefore, we define the notion of strong bisimulation on LPEs directly.

Definition 2.5. Let $\mathcal{A} = (\Sigma, \mathcal{E})$ be a datatype. Let

$$X(d_1{:}D_1, \ldots, d_n{:}D_n) = \sum_{i \in I} \sum_{e^i_1 : E^i_1} \ldots \sum_{e^i_{n_i} : E^i_{n_i}} a_i(f_i)\, X(g^i_1, \ldots, g^i_n) \lhd c_i \rhd \delta$$

and

$$Y(d'_1{:}D'_1, \ldots, d'_{n'}{:}D'_{n'}) = \sum_{i \in I'} \sum_{e'^i_1 : E'^i_1} \ldots \sum_{e'^i_{n'_i} : E'^i_{n'_i}} a'_i(f'_i)\, Y(g'^i_1, \ldots, g'^i_{n'}) \lhd c'_i \rhd \delta$$

be LPEs. Assume $\mathbb{M}$ is a model for $\mathcal{A}$. Let $\mathbf{D}_1, \ldots, \mathbf{D}_n$, $\mathbf{D}'_1, \ldots, \mathbf{D}'_{n'}$, $\mathbf{E}^i_1, \ldots, \mathbf{E}^i_{n_i}$ for all $i \in I$ and $\mathbf{E}'^i_1, \ldots, \mathbf{E}'^i_{n'_i}$ for all $i \in I'$ be the domains belonging to $D_1, \ldots, D_n$, $D'_1, \ldots, D'_{n'}$, $E^i_1, \ldots, E^i_{n_i}$ for all $i \in I$ and $E'^i_1, \ldots, E'^i_{n'_i}$ for all $i \in I'$. We write $\mathbf{d}$ for elements from $\mathbf{D}_1 \times \cdots \times \mathbf{D}_n$ and the jth element of $\mathbf{d}$ is written as $\mathbf{d}_j$. Similarly, we write $\mathbf{d}'$ for elements from $\mathbf{D}'_1 \times \cdots \times \mathbf{D}'_{n'}$, $\mathbf{e}^i$ for elements from $\mathbf{E}^i_1 \times \cdots \times \mathbf{E}^i_{n_i}$ and $\mathbf{e}'^{i'}$ for elements from $\mathbf{E}'^{i'}_1 \times \cdots \times \mathbf{E}'^{i'}_{n'_{i'}}$. Below σ is a valuation mapping variable d_j to $\mathbf{d}_j$ and variable e^i_j to value $\mathbf{e}^i_j$, and σ' is a valuation that maps variable d'_j to $\mathbf{d}'_j$ and variable $e'^{i'}_j$ to value $\mathbf{e}'^{i'}_j$. A relation $R \subseteq (\mathbf{D}_1 \times \cdots \times \mathbf{D}_n) \times (\mathbf{D}'_1 \times \cdots \times \mathbf{D}'_{n'})$ is called a bisimulation relation w.r.t. $\mathcal{A}$ and $\mathbb{M}$ iff for all $\mathbf{d}$ and $\mathbf{d}'$ such that $\mathbf{d} R \mathbf{d}'$

- if for all $i \in I$ and $\mathbf{e}^i$ such that $[\![c_i]\!]^\sigma$, there is some $i' \in I'$ and $\mathbf{e}'^{i'}$ such that $[\![c'_{i'}]\!]^{\sigma'}$, $a_i = a'_{i'}$, $[\![f_i]\!]^\sigma = [\![f'_{i'}]\!]^{\sigma'}$ and $[\![g^i_1, \ldots, g^i_n]\!]^\sigma R [\![g'^{i'}_1, \ldots, g'^{i'}_{n'}]\!]^{\sigma'}$.
- Vice versa.

We say that two terms $X(t_1, \ldots, t_n)$ and $Y(t'_1, \ldots, t'_{n'})$ are strongly bisimilar w.r.t. $\mathcal{A}$, notation $X(t_1, \ldots, t_n) \underline{\leftrightarrow} Y(t'_1, \ldots, t'_{n'})$, iff for all models $\mathbb{M}$ of $\mathcal{A}$ and valuations σ there exists a bisimulation relation R w.r.t. $\mathcal{A}$ and $\mathbb{M}$ such that $[\![t_1, \ldots, t_n]\!]^\sigma R [\![t'_1, \ldots, t'_{n'}]\!]^\sigma$.

3. The μCRL Toolset

The μCRL toolset is a collection of tools manipulating data and process descriptions written in μCRL. The toolset contains four groups of tools (see [17] for a detailed overview of all tools).

Linearizing tools. This group contains one tool, called `mcrl`.The tool `mcrl` transforms μCRL process definitions to linear process equations (LPEs). See [11, 15] for linearisation algorithms and [17] for a detailed description of `mcrl`. All the other tools of the toolset work on LPEs. Thus, before doing any analysis, `mcrl` must be invoked with a μCRL specification as input.

State space explorators. This group contains two tools. The `instantiator` which generates from an LPE a concrete transition system and the simulator, `msim`, which can single-step a process. The instantiator is highly optimized by using a very fast compiling rewriter. The output of the instantiator can be read, manipulated and, visualized by the CADP (Caesar/Aldebaran) toolset [7].

Reduction tools. This group contains four reduction tools (each of them reads an LPE and writes an LPE). These reduction tools are `constelm`, constant elimination, `parelm`, parameter elimination, `sumelm`, sum elimination, and `structelm`, structure elimination. These reduction tools are the implementations of the reduction methods described in this paper.

Rewrite tools. This group contains the tool `rewr`, rewrite, which normalizes the LPE, with respect to the rewrite rules a given abstract data type. If a condition belonging to a summand is equal to false then that summand will be removed.

4. Reduction Methods

4.1 Elimination of Constant Process Parameters

Some data parameters are constant throughout any run of the process. These parameters can be eliminated. All occurrences of these parameters throughout the LPE are replaced by their initial value. This does not reduce the state space of a process, but it may considerably shorten the time to generate a state space from a linear process operator, and can make other reduction tools much more effective.

4.1.1 Algorithm to Detect Constant Parameters

1. Mark all process parameters.
2. Define a substitution ρ that replaces all marked process parameters with its initial value, and that leaves other variables unchanged.
3. If a process parameter d_j with initial value v exists such that $g_j^i\rho \doteq v$ does not rewrite to true, where g_j^i is a process argument that occurs in a summand i with condition c_i, and $c_i\rho$ does not rewrite to false, then d_j must be unmarked and this algorithm must be continued at step 2.
4. Repeat step 2-3 until no additional process parameter becomes unmarked.
5. Remove all marked process parameters from the LPE. Substitute the initial value for all occurrences of marked process parameters in conditions and action arguments.

In the sequel we call this algorithm `constelm`.

Example 4.1. Let D be a data sort, and r and s actions.

$$\begin{aligned}&\texttt{proc } X(a{:}D, b{:}D, c{:}D, d{:}D) = && r(b)\,X(b,a,d,c) \lhd c \doteq 0 \rhd \delta + \\ &&& s(c)\,X(1,b,0,c+d)\\ &\texttt{proc } Y(a{:}D, b{:}D) = r(b)\,Y(b,a) + s(0)\,Y(1,b)\end{aligned}$$

`Constelm` finds that $X(0,0,0,0) \underline{\leftrightarrow} Y(0,0)$.

Note that the complexity of the algorithm is $O(mN)$ with m the number of process parameters, and N the size of the input LPE, assuming that rewriting takes constant time.

Theorem 4.2. Consider the LPEs as in Definition 2.5, with respective initial states $\vec{x}$ and $\vec{y}$. Assume that the LPE for Y with initial state $\vec{y}$ has been obtained from the LPE for X with initial state $\vec{x}$ by applying `constelm`. Then $X(\vec{x}) \underline{\leftrightarrow} Y(\vec{y})$.

4.2 Elimination of Dead Process Parameters

Some process parameters and sum variables do not influence the behaviour of a system, as they do not occur directly or indirectly in conditions and actions. By removing these parameters, the state space of an LPE can be reduced considerably.

4.2.1 Algorithm to Mark Parameters as Influential Parameters.

We mark parameters if they can have an influence on the behaviour of the LPE. The remaining parameters can be removed. Here follows the algorithm, which we call `parelm`.

1. Mark the process parameters which occur in the conditions and in the action arguments of the LPE. These are not removed, as they clearly have influence on the behaviour of the process.
2. If a process parameter d_j is marked then mark also all process parameters occurring in g_j^i for all $i \in I$, as these parameters can indirectly influence the process behaviour. Repeat this step until no more parameters are marked.
3. Remove all unmarked parameters from the process.
4. Remove for each summand all those sum variables which do neither occur in the condition, nor in the arguments of the action, nor in the arguments of the new state.

Example 4.3. Assume X and Y are defined by:

$$\begin{aligned} \texttt{proc}\ X(a{:}D, b{:}D, c{:}D) &= s\, X(a,c,b) + \sum_{d:D} r(c)\, X(d,b,c) \\ \texttt{proc}\ Y(b{:}D, c{:}D) &= s\, Y(c,b) + r(c)\, Y(b,c) \end{aligned}$$

As parameter a has no influence on the behaviour of process X, `parelm` finds for all a, b and c that $X(a,b,c) \underline{\leftrightarrow} Y(b,c)$.

Theorem 4.4. Consider the LPEs as in Definition 2.5, with respective initial states $\vec{x}$ and $\vec{y}$. Assume that the LPE for Y with initial state $\vec{y}$ has been obtained from the LPE for X with initial state $\vec{x}$ by applying `parelm`. Then $X(\vec{x}) \underline{\leftrightarrow} Y(\vec{y})$.

4.3 Elimination of Sum Variables

There are cases in which a condition forces a sum variable to be equal to one particular data term. A transformation step towards simpler LPEs can be made by eliminating these sum variables and replacing their occurrences by that value. We call this transformation sum elimination, or `sumelm`.

A condition and a sum variable of a summand determine a set of values. Each value of is a candidate to be substituted in all occurrences of that sum variable in that summand.

4.3.1 Computing the set of candidates.

The function $\mathit{Candidates}(x, c)$, in which x is a sum variable and c a condition, computes a set of candidates for replacement of sum variable x. The function is defined by induction.

$$\begin{aligned} \mathit{Candidates}(x, v \doteq w) &= \begin{cases} \text{if } x \equiv v \wedge x \notin w & \text{then } \{w\} \\ \text{if } x \equiv w \wedge x \notin v & \text{then } \{v\} \\ \emptyset, \text{ otherwise} \end{cases} \\ \mathit{Candidates}(x, (c_1 \wedge c_2)) &= \mathit{Candidates}(x, c_1) \cup \mathit{Candidates}(x, c_2) \\ \mathit{Candidates}(x, (c_1 \vee c_2)) &= \mathit{Candidates}(x, c_1) \cap \mathit{Candidates}(x, c_2) \\ \mathit{Candidates}(x, c) &= \emptyset, \text{ otherwise} \end{aligned}$$

where x stands for variables, v and w stand for data terms and c, c_1 and c_2 stand for conditions. The expression $x \in v$ stands for x occurs in v. Note that in order to calculate the intersection of two sets, it must be determined that values are distinct, which requires the use of $\doteq$ and rewriting.

4.3.2 Substitution and Elimination.

For all $i \in I$, $j \in 1 \ldots n_i$, at which the set $\mathit{Candidates}(e^i_j, c_i)$ is not empty (choose a value v from this set), or the sort of e^i_j contains exactly one element (call that element v):

1. Substitute v in all occurrences of e^i_j in the ith summand of the LPE, and
2. Eliminate e^i_j from the list of sum variables of the ith summand.

Example 4.5. Consider

$$\begin{aligned}\texttt{proc}\ X(d{:}\mathbf{Bool}) &= \sum_{b:\mathbf{Bool}} a(b).X(b) \lhd b \doteq \mathsf{f} \rhd \delta\\ \texttt{proc}\ Y(d{:}\mathbf{Bool}) &= a(\mathsf{f}).Y(\mathsf{f}) \lhd \mathsf{f} \doteq \mathsf{f} \rhd \delta\end{aligned}$$

`Sumelm` yields $X(d) \underline{\leftrightarrow} Y(d)$. Rewriting the condition in Y makes it equal to `t` (true).

Theorem 4.6. Consider the LPEs as in Definition 2.5. Assume that the LPE for Y has been obtained from the LPE for X by applying `sumelm`. Then, for state vector $\vec{x}$ it holds that $X(\vec{x}) \underline{\leftrightarrow} Y(\vec{x})$.

4.4 An application of the reduction tools to an example

To demonstrate the cooperation of the three reduction algorithms, we take a slightly more substantial example. The patterns which are found in this example occur often in bigger specifications after linearization.

sort $\mathbf{Bool}, \mathbf{Bit}, \mathbf{D}$
func $\mathsf{t}, \mathsf{f}: \to \mathbf{Bool}$, $d_1, d_2 :\to \mathbf{D}$, $0, 1: \to \mathbf{Bit}$
map $\neg : \mathbf{Bool} \to \mathbf{Bool}$, $\vee : \mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool}$
$\doteq{:}\mathbf{D} \times \mathbf{D} \to \mathbf{Bool}$, $\doteq{:}\mathbf{Bit} \times \mathbf{Bit} \to \mathbf{Bool}$
var $b : \mathbf{Bool}$
rew $\neg\mathsf{t} = \mathsf{f}$, $\neg\mathsf{f} = \mathsf{t}$, $\mathsf{t} \vee b = \mathsf{t}$, $\mathsf{f} \vee b = b$
var $d : \mathbf{D}$
rew $d_1 \doteq d_2 = \mathsf{f}$, $d_2 \doteq d_1 = \mathsf{f}$, $d \doteq d = \mathsf{t}$
var $b : \mathbf{Bit}$
rew $0 \doteq 1 = \mathsf{f}$, $1 \doteq 0 = \mathsf{f}$, $b \doteq b = \mathsf{t}$
proc $X(d{:}\mathbf{D}, b{:}\mathbf{Bit}) = \sum_{d_0:\mathbf{D}} \tau\, X(d_0, b) \lhd d \doteq d_2 \vee b \doteq 0 \rhd \delta + \sum_{b_0:\mathbf{Bit}} \tau\, X(d, b_0) \lhd b_0 \doteq 0 \rhd \delta$
init $X(d_1, 0)$

- Algorithm `parelm` does not change the LPE, because the process parameters d and b occur in one of the conditions.
- Algorithm `constelm` does not change the LPE, because the sum variables d_0 or b_0 occur in both process argument vectors.
- Algorithm `sumelm` changes the LPE. It substitutes 0 in each occurrence of sum variable b_0 and removes that sum variable. The equation becomes

  proc $X(d{:}\mathbf{D}, b{:}\mathbf{Bit}) = \sum_{d_0:\mathbf{D}} \tau\, X(d_0, b) \lhd d \doteq d_2 \vee b \doteq 0 \rhd \delta + \tau\, X(d, 0) \lhd 0 \doteq 0 \rhd \delta$
  init $X(d_1, 0)$

- Algorithm `parelm` still cannot change the modified LPE.
- Algorithm `constelm` changes the LPE. It substitutes 0 in each occurrence of b and removes that parameter. The equation becomes

$$\begin{array}{ll}\textbf{proc} & X(d{:}\mathbf{D}) = \sum_{d_0:\mathbf{D}} \tau\, X(d_0) \lhd d \doteq d_2 \vee 0 \doteq 0 \rhd \delta + \tau\, X(d) \lhd 0 \doteq 0 \rhd \delta \\ \textbf{init} & X(d_1)\end{array}$$

- After rewriting the equation becomes

$$\begin{array}{ll}\textbf{proc} & X(d{:}\mathbf{D}) = \sum_{d_0:\mathbf{D}} \tau\, X(d_0) \lhd \mathtt{t} \rhd \delta + \tau\, X(d) \lhd \mathtt{t} \rhd \delta \\ \textbf{init} & X(d_1)\end{array}$$

- Algorithm `parelm` changes the LPE. Process parameter d does not occur in the condition. Process parameter d will be removed. Sum variable d_0 does not occur in the equation. Sum variable d_0 will be removed.

$$\begin{array}{ll}\textbf{proc} & X() = \tau\, X() \lhd \mathtt{t} \rhd \delta + \tau\, X() \lhd \mathtt{t} \rhd \delta \\ \textbf{init} & X()\end{array}$$

The state space is reduced from two states to one state, a reduction of 50 per cent.

4.5 Elimination of Structured Variables

There are possible reductions of an LPE which are not detected by the previously mentioned reduction methods because the required properties of the variables are hidden within complex data terms. We describe here an expansion method, which replaces terms with a constructor function as head symbols by the name of the constructor and its arguments. In this way variables occuring in subterms are better exposed and more amenable to be eliminated by one of the other tools. 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. We call this expansion method `structelm`, short for structure elimination.

4.5.1 Unfolding a Structured Sort S.

1. Detect how many constructors generate $\mathbf{S}$. Assume that $\mathbf{S}$ is generated by k constructors $f_i{:}\ \mathbf{D}_i^1 \times \cdots \times \mathbf{D}_i^{m_i} \to \mathbf{S}$, with $1 \leq i \leq k$.

2. Add a new enumerated sort $\mathbf{U}_k$ with k constant constructors, $c_1, \ldots, c_k$, to the abstract data type. The elements of $\mathbf{U}_k$ represent each a different constructor of sort $\mathbf{S}$.

3. Add a function $C{:}\ \mathbf{U}_k \times \mathbf{S} \times \cdots \times \mathbf{S} \to \mathbf{S}$ to the abstract data type with arity $k + 1$. Add the following rewrite rules to the abstract data type.

   $$\begin{array}{ll}\textbf{rew} & C(c_i, x_1, \ldots, x_i, \ldots, x_k) = x_i \\ & C(e, x \ldots, x) = x\end{array}$$

   These functions C are called *case functions*. Note that strictly spoken the last rewrite rule is not necessary, but it sometimes turns out to be useful.

4. Add projection functions $det{:}\ \mathbf{S} \to \mathbf{U}_k$ and $\pi_i^j{:}\ \mathbf{S} \to \mathbf{D}_i^j$ with $1 \leq j \leq m_i$ and $1 \leq i \leq k$ to the abstract data type. Add the following rewrite rules to the abstract data type.

   $$\begin{array}{ll}\textbf{rew} & det(f_i(d_i^1, \ldots, d_i^j, \ldots, d_i^{m_i})) = c_i \\ & \pi_i^j(f_i(d_i^1, \ldots, d_i^j, \ldots, d_i^{m_i})) = d_i^j\end{array}$$

The mapping *det* gives the encoding for the head constructor of a term. The mapping π_i^j gives the jth subterm of a term with head constructor represented by c_i. Note that for the functions *det* to exist, it must hold that for all $1 \leq i < j \leq k$ and all $d_i^1, \ldots, d_i^{m_i}, d_j^1, \ldots, d_j^{m_j}$ it is the case $f_i(d_i^1, \ldots, d_i^{m_i}) \neq f_j(d_j^1, \ldots, d_j^{m_j})$. For the functions π_i^j to exist it must hold that for all $d^1, \ldots, d^j, e^j, d^{j+1}, \ldots, d^{m_i}$ with $d^j \neq e^j$, it is the case that $f_i(d^1, \ldots, d^j, \ldots, d^{m_i}) \neq f_i(d^1, \ldots, e^j, \ldots, d^{m_i})$.

5. Add the following rewrite rules for all constructors and maps $f\colon \mathbf{D}_1 \times \ldots \times \mathbf{D}_n \to \mathbf{S}$ (except for the just added case function C, but including *det* and π_i^j) of the abstract data type:

   rew $f(x_1, \ldots, C(e, y_1, \ldots, y_m), \ldots x_n) =$
   $C(e, f(x_1, \ldots, y_1, \ldots, x_n), \ldots, f(x_1, \ldots, y_m, \ldots, x_n))$

   Note that the case function C in the right hand side can be ill-typed. Then a correctly typed case function with the rewrite rules of step 3 and this step must be added. The rewrite rules added here are actually derivable from the rewrite rules in step 3 using the fact that $\mathbf{U}_k$ contains exactly k elements. These rules are however essential to clean up an LPE after `structelm` has been applied to it, which is needed for the other tools to proceed.

6. Replace the declaration of each process parameter of sort $\mathbf{S}$ by the declarations belonging to the unfolded process parameter: `proc` $X(\ldots, d\colon \mathbf{S}, \ldots)$ becomes:

   `proc` $X(\ldots, e{:}\mathbf{U}_k, d_1^1{:}\mathbf{D}_1^1, \ldots, d_1^{m_1}{:}\mathbf{D}_1^{m_1}, d_2^1{:}\mathbf{D}_2^1, \ldots, d_k^{m_k}{:}\mathbf{D}_k^{m_k}, \ldots)$.

7. Replace the declaration of each sum variable of sort $\mathbf{S}$ by the declarations of the sum variables belonging the unfolded sum variable. $\sum_{d:\mathbf{S}}$ becomes $\sum_{e:U_k} \sum_{d_1^1:\mathbf{D}_1^1} \cdots \sum_{d_1^{m_1}:\mathbf{D}_1^{m_1}} \sum_{d_2^1:\mathbf{D}_2^1} \cdots \sum_{d_k^{m_k}:\mathbf{D}_k^{m_k}}$.

8. Replace each occurrence of a structured parameter or sum variable d of sort $\mathbf{S}$ in the LPE by $C(e, f_1(d_1^1, \ldots, d_1^{m_1}), \ldots, f_k(d_k^1, \ldots, d_k^{m_k}))$.

9. Translate a process argument g_j^i of sort $\mathbf{S}$ to its unfolded version. Concretely, $X(\ldots, g_j^i, \ldots)$ becomes

   $X(\ldots, det(g_j^i), \pi_1^1(g_j^i), \ldots, \pi_1^{m_1}(g_j^i), \pi_2^1(g_j^i), \ldots, \pi_k^{m_k}(g_j^i), \ldots)$.

Theorem 4.7. Consider the LPEs as in Definition 2.5, with respective initial states $\vec{x}$ and $\vec{y}$. Assume that the LPE for Y with initial state $\vec{y}$ has been obtained from the LPE for X with initial state $\vec{x}$ by applying `structelm` to a constructor sort $\mathbf{S}$. Assume also that all functions *det* exist (see Section 4.5.1 item 4). Then $X(\vec{x}) \underline{\leftrightarrow} Y(\vec{y})$.

4.6 The example revisited

The example in section 4.4 is slightly changed to show the effect of `structelm`. The difference is that sort **D** and sort **Bit** are encapsulated in a new sort **Frame**. The modified example is strongly bisimilar to the original one. We extend the previous example with the following data type.

sort **Frame**
func *frame*: $\mathbf{D} \times \mathbf{Bit} \to \mathbf{Frame}$, *void*: $\to \mathbf{Frame}$
map *data*: $\mathbf{Frame} \to \mathbf{D}$, *bit*: $\mathbf{Frame} \to \mathbf{Bit}$
var $d{:}\mathbf{D}, b{:}\mathbf{Bit}$
rew $data(frame(d, b)) = d$, $bit(frame(d, b)) = b$
$data(void) = d_1$, $bit(void) = 0$

We replace the previous process equation with

$$\begin{array}{ll}\textbf{proc} & X(f{:}\textbf{Frame}) \\ & = \sum_{d_0:\textbf{D}} \tau\, X(\mathit{frame}(d_0, \mathit{bit}(f))) \lhd \mathit{data}(f) \doteq d_2 \vee \mathit{bit}(f) \doteq 0 \rhd \delta \\ & + \sum_{b_0:\textbf{Bit}} \tau\, X(\mathit{frame}(\mathit{data}(f), b_0)) \lhd b_0 \doteq 0 \rhd \delta \\ \textbf{init} & X(\mathit{frame}(d_1, 0)).\end{array}$$

`constelm` and `parelm` do not change the LPE. Before `constelm` and `parelm` will do the job, the process parameter f of sort **Frame** must be unfolded. We have already an enumerated sort of size 2, namely **Bool**, which we can use instead of $\mathbf{U}_2$. In addition to the projections *bit* and *data* which already exist, a map **Frame** $\to$ **Bool** must also be defined which determines which function symbol is the header of a term of sort **Frame**, *void* or *frame*. We call this map *det*. Before expanding the LPE the following maps and rewrite rules must be added.

$$\begin{array}{ll}\textbf{map} & C : \textbf{Bool} \times \textbf{Frame} \times \textbf{Frame} \to \textbf{Frame} \\ & C : \textbf{Bool} \times \textbf{Bit} \times \textbf{Bit} \to \textbf{Bit} \\ & C : \textbf{Bool} \times \mathbf{D} \times \mathbf{D} \to \mathbf{D} \\ & \mathit{det} : \textbf{Frame} \to \textbf{Bool} \\ \textbf{var} & f_1, f_2{:}\textbf{Frame} \\ \textbf{rew} & C(\mathsf{t}, f_1, f_2) = f_1,\ C(\mathsf{f}, f_1, f_2) = f_2 \\ \textbf{var} & d{:}\mathbf{D}, b{:}\textbf{Bit} \\ \textbf{rew} & \mathit{det}(\mathit{void}) = \mathsf{t},\ \mathit{det}(\mathit{frame}(d, b)) = \mathsf{f}\end{array}$$

Besides the distributive rules, these rewrite rules are sufficient for this example. The expanded LPE becomes

$$\begin{array}{ll}\textbf{proc} & X(e{:}\textbf{Bool}, d{:}\mathbf{D}, b{:}\textbf{Bit}) \\ & = \sum_{d_0:\textbf{D}} \tau\, X(\\ & \mathit{det}(\mathit{frame}(d_0, \mathit{bit}(C(e, \mathit{void}, \mathit{frame}(d, b))))), \\ & \mathit{data}(\mathit{frame}(d_0, \mathit{bit}(C(e, \mathit{void}, \mathit{frame}(d, b))))), \\ & \mathit{bit}(\mathit{frame}(d_0, \mathit{bit}(C(e, \mathit{void}, \mathit{frame}(d, b)))))) \\ & \lhd \mathit{data}(C(e, \mathit{void}, \mathit{frame}(d, b))) \doteq d_2 \vee \mathit{bit}(C(e, \mathit{void}, \mathit{frame}(d, b))) \doteq 0 \rhd \delta \\ & + \sum_{b_0:\textbf{Bit}} \tau\, X(\\ & \mathit{det}(\mathit{frame}(\mathit{data}(C(e, \mathit{void}, \mathit{frame}(d, b))), b_0)), \\ & \mathit{data}(\mathit{frame}(\mathit{data}(C(e, \mathit{void}, \mathit{frame}(d, b))), b_0)), \\ & \mathit{bit}(\mathit{frame}(\mathit{data}(C(e, \mathit{void}, \mathit{frame}(d, b))), b_0))) \\ & \lhd b_0 \doteq 0 \rhd \delta \\ \textbf{init} & X(\mathsf{f}, d_1, 0).\end{array}$$

After rewriting we get

$$\begin{array}{ll}\textbf{proc} & X(e{:}\textbf{Bool}, d{:}\mathbf{D}, b{:}\textbf{Bit}) \\ & = \sum_{d_0:\textbf{D}} \tau\, X(\mathsf{t}, d_0, C(e, 0, b)) \lhd C(e, d_1, d) \doteq d_2 \vee C(e, 0, b) \doteq 0 \rhd \delta \\ & + \sum_{b_0:\textbf{Bit}} \tau\, X(\mathsf{t}, C(e, d_1, d), b_0) \lhd b_0 \doteq 0 \rhd \delta \\ \textbf{init} & X(\mathsf{f}, d_1, 0).\end{array}$$

By applying `constelm` we find that $e = \mathsf{f}$. The process then reduces to the process in example 4.4.

5. Cooperation and influence of reductions

The different tools interact. In particular it might be that tool `A` does not lead to a reduction, but application of tool `B` may make application of tool `A` fruitful again. We identified the cases where

this is the case and put these in Table 1. Horizontally, the tools can be found that do not lead to a

	`structelm`	`sumelm`	`parelm`	`constelm`
`structelm`	**X**	Yes	Yes	Yes
`sumelm`	No	**X**	Yes	Yes
`parelm`	No	No	**X**	No
`constelm`	No	Yes	Yes	**X**

Table 1: Dependence of tools

reduction, but may become effective after applying one of the tools put vertically. So, for instance, the table indicates that if `structelm` is applied when `sumelm` does not yield any reduction, then applying `sumelm` after `structelm` might yield additional reductions. We put **X**'s on the diagonal, because these positions have no meaning.

Note that iteratively applying the same tool is only useful for `structelm`. But if there are no recursive types, then only a finite number of applications of `structelm` can take place.

6. Experimental results

Bounded retransmission protocol						
	before reduction			after reduction		
l	# states	# transitions	time	# states	# transitions	time
1	454	518	1.34s	454	518	1.45s
2	1856	2134	2.02s	834	954	1.65s
3	10550	12170	6.28s	1202	1378	1.89s
4	86968	100406	44.09s	2224	2558	2.49s
5	968538	1118498	8m27.13s	8642	9970	6.45s
Onebit sliding window protocol						
	before reduction			after reduction		
n	# states	# transitions	time	# states	# transitions	time
1	39577	229650	21.19s	39577	229650	20.42s
2	319912	1937388	3m44.29s	39577	229650	19.37s
3	1208737	7484714	29m56.95s	39577	229650	18.79s
Firewire link layer protocol						
	before reduction			after reduction		
n	# states	# transitions	time	# states	# transitions	time
1	95160	158017	17m15.58s	74271	123370	59m31.00s
2	371804	641565	1h9m15.30s	74271	145456	1h0m1.79s
3	872224	1548401	2h22m18.41s	74271	167542	59m56.56s

Table 2: The effect of the elimination tools on the size of state spaces

We applied our reduction techniques to a number of data transfer protocols that have been described in μCRL. These are the bounded retransmission protocol [9], the one bit sliding window protocol [3] and the Firewire link layer protocol [13]. These protocols are quite different in nature. For all protocols we hid the delivery of data, in order to investigate the control structure. By applying the different

elimination algorithms we were able to remove most or even all occurrences of variables refering to data reducing the state space substantially. If all variables refering to data are removed, results on the control structure hold for any data domain, in particular for those of infinite size.

In the tables we list the sizes of the state spaces for different sizes of the data domain. For the bounded retransmission protocol the length of the data packages and the number of data elements is given by l and the number of retransmissions has been set to 3. For the two other protocols the number of data elements is given by n. The one bit sliding window protocol is a unidirectional sliding window protocol with buffer size 1 at both the sending and receiving sides. The Firewire link layer protocol consists of a bus and two link processes. The results have been obtained on a SGI Powerchallenge with R12000 processors on 300Mhz. The version of the toolset that has been used is 2.8.3.

6.0.1 Acknowledgments.

We thank Jaco van de Pol for his comments on this paper.

References

1. M.G.J. van den Brand, P. Klint, and P.A. Olivier. Compilation and Memory Management for ASF+SDF. In: Proceedings in Compiler Construction (CC'99) (ed. S. Jähnichen), LNCS 1575, 198–213, Springer-Verlag, 1999.
2. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Communication*, 60(1/3):109-137,1984.
3. M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. *The Computer Journal*, 37(4): 289-307, 1994.
4. S.C.C. Blom. Verification of the Euris railway control specification of Woerden-Harmelen. Work in progress. Centrum voor Wiskunde en Informatica, Amsterdam, 2000
5. S.C.C. Blom. Partial τ-confluence for efficient state space generation. Technical report, CWI, 2001; to appear.
6. W.J. Fokkink. Introduction to Process Algebra, In Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, 2000.
7. H. Garavel. `OPEN/CAESAR`: An Open Software Architecture for Verification, Simulation, and Testing". In G. Goos, J. Hartmanis, and J. van Leeuwen, Editors, *Proceedings of TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, Springer Verlag, pages 68-84, 1998. Available from `http://www.inrialpes.fr/vasy/Publications.`
8. J.F. Groote. The syntax and semantics of timed μCRL. Report SEN-R9709. CWI, The Netherlands, 1997.
9. J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. A case study in computer checked verification. In M. Wirsing and M. Nivat, Editors, *Proceedings of AMAST'96*, Munich, volume 1101 of *Lecture Notes in Computer Science*, Springer Verlag, pages 536-550, 1996.
10. J.F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In A. Ponse and C. Verhoef and S.F.M. van Vlijmen,editors, *Workshops in Computing*, pages 26–62. Springer-Verlag, 1994.
11. J.F. Groote, A. Ponse and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1-2):39-72, 2001.
12. J.F. Groote and M.P.A. Sellink. Confluence for Process Verification. In *Theoretical Computer Science B (Logic, semantics and theory of programming)*, 170(1-2):47-81, 1996.

13. S.P. Luttik. Description and formal specification of the link layer of P1394. Technical Report SEN-R9706, CWI, Amsterdam, 1997. ftp://ftp.cwi.nl/pub/CWIreports/SEN/SEN-R9706.ps.Z
14. J.C. van de Pol. A Prover for the μCRL Toolset with Applications – version 0.1. Technical report SEN-R0106, CWI, Amsterdam, 2001.
15. Y.S. Usenko. Linearization of μCRL processes. To appear. 2001.
16. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume II, pages 677–788. Elsevier Science Publishers B.V., 1990.
17. A.G. Wouters Manual for the μCRL toolset (version 2.07). Technical report, CWI, 2001; to appear. Available at http://www.cwi.nl/∼mcrl

A. The μCRL Toolset runs the Example

We show how we applied to μCRL toolset to the running example.

```
sort D
func d1,d2: -> D
map  eq:D#D -> Bool
rew  eq(d1,d1)=T eq(d2,d2)=T eq(d1,d2)=F eq(d2,d1)=F

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

var  x:Bool
rew  and(T,x)=x and(x,T)=x and(x,F)=F and(F,x)=F
     or(T,x)=T  or(x,T)=T  or(x,F)=x or(F,x)=x
     not(F)=T not(T)=F
     eq(x,T)=x eq(T,x)=x eq(F,x)=not(x) eq(x,F)=not(x)

sort Bit
func 0,1:-> Bit
map  eq:Bit#Bit -> Bool
rew  eq(0,0)=T eq(1,1)=T eq(0,1)=F eq(1,0)=F

sort Frame
func frame:D#Bit -> Frame
     void:->Frame
map  data:Frame -> D
     bit:Frame -> Bit
var  d:D b:Bit
rew  data(void)=d1 data(frame(d,b)) =d
     bit(void)=0 bit(frame(d,b))=b

proc X(f:Frame) = sum(d0:D,tau.X(frame(d0, bit(f)))
                  <|or(eq(data(f),d2), eq(bit(f),0))|>delta)
                + sum(b0:Bit,tau.X(frame(data(f),b0))<|eq(b0,0)|>delta)

init X(frame(d1,0))
```

Most μCRL commands [17] are filters which read from `stdin` and write to `stdout`, so they can be used in a pipeline. The reduced LPE is written to the file `output`. Internally LPEs are input and output in a compressed format, called the tbf format. The pretty printer, `pp`, is used to transform tbf files to a readable form. Below the command is given that transforms the example, followed by its output.

```
mcrl -stdout -regular -nocluster input|structelm -expand Frame|rewr -case|   \
sumelm|constelm|parelm|pp > output
```

```
Structelm: Generated casefunction: C2-Frame(Bool,Frame,Frame)
Structelm: Parameter f: "Frame" is expanded to 3 parameters
Structelm: Number of parameters of input LPE: 1
Structelm: Number of parameter(s) of output LPE:3
Rewr: structelm -case-completion has ended successfully
Rewr: NOW COMPILING
Rewr: gcc -O4 -DNDEBUG -c REWRITERALT.c -o REWRITER.o
Rewr: COMPILING FINISHED
Rewr: gcc -shared REWRITER.o -o REWRITER.so
Rewr: REWRITER LOADED
Rewr: Read 2 summands
Rewr: Written 2 summands
Sumelm: NOW COMPILING
Sumelm: gcc -O4 -DNDEBUG -c REWRITERALT.c -o REWRITER.o
Sumelm: COMPILING FINISHED
Sumelm: gcc -shared REWRITER.o -o REWRITER.so
Sumelm: REWRITER LOADED
Sumelm: Read 2 summands
Sumelm: Number of parameters of input LPE: 3
Sumelm: Summand 2: substituted for "b0" the term 0
Sumelm: In this summand is (are) 1 variable(s) eliminated
Sumelm: Written 2 summands
Sumelm: Number of parameters of output LPE: 3
Constelm: NOW COMPILING
Constelm: gcc -O4 -DNDEBUG -c REWRITERALT.c -o REWRITER.o
Constelm: COMPILING FINISHED
Constelm: gcc -shared REWRITER.o -o REWRITER.so
Constelm: REWRITER LOADED
Constelm: Read 2 summands
Constelm: Number of parameters of input LPE: 3
Constelm: Constant parameter "f_e" (pos 1, sort "Bool") = "F" is removed
Constelm: Constant parameter "f_1" (pos 3, sort "Bit") = "0" is removed
Constelm: Number of removed parameter(s):2
Constelm: Number of remaining parameter(s) of output LPE:1
Constelm: Written 2 summands
Constelm: Number of parameters of output LPE: 1
Parelm: Number of parameters of input LPE: 1
Parelm: Parameter "f_0" (sort "D") is removed
Parelm: In total 1 process parameter has been removed
Parelm: Variable "d0" (sort "D") in the sum operator of summand 1 has been deleted
Parelm: Number of parameters of output LPE: 0
```

The computed reduced LPE can be found in the file `output` and looks as follows.

```
proc X =
tau. X <|T|>delta+
tau. X <|T|>delta
init X
```