Proof Theory for μCRL

Jan Friso Groote
Alban Ponse
Department of Software Technology, CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

A proof theory for the specification language μCRL (*micro* CRL) is proposed. μCRL consists of process algebra extended with abstract data types. The proof theory is meant to formalize the interaction between processes and data. Furthermore it provides the means to prove properties about these in a precise way. The proof theory has been designed such that automatic proof checking is feasible.

A simple language is defined in which basic properties of processes and of data can be expressed. A proof system is presented for this property language, comprising a rule for induction, the Recursive Specification Principle, and process algebra axioms. The proof theory is illustrated with small examples, and a case study about a bag.

Key Words & Phrases: Proof theory for specification language, ADT (Abstract Data Types), Process Algebra.
1985 Mathematics Subject Classification: 68Q99.
1987 CR Categories: D.2.4, D.3.1, D.3.3, F3.1.
Note: The authors are supported by the European Communities under RACE project no. 1046, Specification and Programming Environment for Communication Software (SPECS). The first author is also supported by ESPRIT Basic Research Action 3006 (CONCUR). This document does not necessarily reflect the view of the SPECS project.

1 Introduction

In this paper we provide the simple, algebraic specification language μCRL with a proof theory. The acronym μCRL stands for *micro* Common Representation Language [GP90, GP91]. This language has been developed under the assumption that an extensive and mathematically precise study of the basic constructs of specification languages will yield fundamental insights that are essential to an analytical approach of much richer (and more complicated) specification languages such as SDL [CCI87], LOTOS [ISO87], PSF [MV90] and CRL [Ss90].

The language μCRL offers a uniform framework for the specification of data and processes. Data is specified by equational specifications: one can declare sorts and functions working upon these sorts, and describe the meaning of these functions by equational axioms. Processes are described in the style of CCS [Mil89], CSP [Hoa85] or ACP [BK84b, BW90], where the particular process syntax has been taken from ACP. In section 2 we give a short overview of the syntax and semantics of μCRL.

The proof theory serves two purposes. First it allows to formalize the interaction between processes and data. Particularly, we can express how the correctness of a protocol depends on characteristics of data. Furthermore it reveals typical characteristics of the data/process relationship. The conditional (or if-then-else) construct is characterised by two simple axioms, relating the standard sort of the Booleans to processes. The axioms for the communication merge reflect that actions may be parameterised with data. The data dependency of processes is captured by an adaptation of the process algebra rule RSP (Recursive Specification Principle). The last place where data and processes meet is the generalised sum construct. It turns out to be a difficult construct with the flavour of universal quantification.

A second purpose is to enable precise proofs of the correctness of concurrent systems and programs. It is well-known that even the slightest error in a program may have serious consequences. Generally advocated techniques such as formal specification and systematic testing reduce the number of mistakes, but they do in no way guarantee overall correctness. We present a proof system that allows for automatic proof checking. The reason for this is that even formal proofs are error prone. If proofs are automatically checked, one may expect a considerably higher degree of correctness. We believe that this is one of the few ways, if not the only one, to deliver error free programmed systems.

In this paper we first define a language in which we can express simple properties of specifications. These properties consist of identities between data or process terms, linked together with propositional connectives $\neg, \vee, \wedge, \rightarrow$ and $\leftrightarrow$. We define a proof system in a natural deduction format because this is close to intuitive reasoning. It contains so called 'logical' axioms and rules, suitable to derive the fundamental properties induced by $=$ and the propositional connectives. Next we introduce 'modules', i.e. sets of axioms and rules, expressing basic identities about data or processes. For instance, the module BOOL contains two axioms. One expressing that *true* and *false* are not equal and another saying that *true* and *false* represent the only Booleans. Another module about data contains an induction rule for many-sorted abstract data specifications. For processes we incorporate adapted versions of standard process algebra modules [BW90].

We believe that μCRL and its proof theory do not have their counterparts in existing formalisms in this area. Among these formalisms we find Hoare logics [Apt81, Apt84], programs as predicate transformers [DS90], UNITY [CM88], I/O automata [LT89], process algebra [Hoa85, Mil89, BW90]. The first three approaches are basically about state transformations and do not concern observable behaviour. As these approaches are essentially about assigning values to variables, the corresponding proof systems also deal with data.

I/O automata are suitable for modelling concurrent and distributed systems, the components of which are (data parameterised) automata describing explicitly the interaction with their environment. Correctness is proved by assigning properties to the states, and using invariance techniques. Contrary to process algebras, I/O automata do not seem to be well-suited for algebraic manipulation.

Traditionally, process algebras do not concentrate on data. There is a large body of theory to prove preorders and equivalences based on observable behaviour. The language μCRL and the proof theory described here are also in this style, but incorporate an explicit notion of data. Two other extensions of process algebra with data are mobile processes [MPW89] and the language *VPL* [HI90]. Mobile processes incorporate data by describing it in a process like way. Data is modelled by pointer structures that can dynamically change. This approach differs structurally from μCRL. The work of [HI90] is in the same vein as μCRL as far as the data in processes is concerned. It is not determined how data itself should be specified.

Acknowledgements. We thank Jos Baeten, Jan Bergstra, Javier Blanco, Henri Korver, Karst Koymans and Piet Rodenburg for helpful comments.

Contents

2 Overview of the language μCRL

This section provides a compact introduction to the language μCRL. For the formal definition of the syntax and semantics of μCRL we refer to [GP90].

2.1 The syntax of μCRL.

First, we assume the existence of a set $\mathcal{N}$ of *names* that are used to denote sorts, variables, functions, processes and labels of actions. The names in $\mathcal{N}$ are words over an alphabet not containing

$$\bot, +, \|, \mathbin{|\!\!\lfloor}, |, \lhd, \rhd, \cdot, \delta, \tau, \partial, \rho, \Sigma, \surd, \times, \rightarrow, :, =,), (, \}, \{, , \text{ a space and a newline.}$$

The space and the newline serve as separators between names and are used for the layout of specifications. The other symbols have special functions. Moreover, $\mathcal{N}$ does not contain the reserved keywords **sort**, **proc**, **var**, **act**, **func**, **comm**, **rew** and **from**.

Data types are specified as the standard abstract data types [EM85], using sorts, functions and axioms. Sorts are declared using the keyword **sort** and functions are declared using the keyword **func**. Axioms are declared using the keyword **rew**, referring to the possibility to use rewriting technology for evaluation of abstract data types. The variables that are used in the axioms must be declared directly before the axioms. Their scope only extends to the next single **rew** declaration.

As an example we define the Booleans. The Booleans must be included in each μCRL specification.

$$\begin{array}{ll} \textbf{sort} & \textbf{Bool} \\ \textbf{func} & T, F :\to \textbf{Bool} \end{array}$$

The following example shows how natural numbers with a zero, a successor, addition and multiplication can be declared.

Example 2.1.1.

$$\begin{array}{ll} \textbf{sort} & Nat \\ \textbf{func} & 0 :\to Nat \\ & S : Nat \to Nat \\ & add, times : Nat \times Nat \to Nat \\ \textbf{var} & x, y : Nat \\ \textbf{rew} & add(x, 0) = x \\ & add(x, S(y)) = S(add(x, y)) \\ & times(x, 0) = 0 \\ & times(x, S(y)) = add(x, times(x, y)) \end{array}$$

(End example.)

Processes may contain actions representing elementary activities that can be performed. These actions must be explicitly declared using the keyword **act**. Actions may be parameterised by data. In the following lines an action declaration is displayed.

$$\begin{array}{ll} \textbf{act} & a, b, c \\ & a, d : Nat \end{array}$$

Here parameterless actions a, b, c and actions a, d depending on natural numbers are declared. Note that overloading is allowed, as long as this cannot lead to confusion (see [GP90] for details). In this case the actions a and $a(n)$ (with n of sort Nat) are different actions.

In μCRL parallel processes communicate via synchronisation of actions. A communication specification, declared using the keyword **comm**, prescribes which actions may synchronise on the level of the labels of actions. For instance, in

$$\textbf{comm} \quad in \,|\, out = com$$

each action $in(t_1, ..., t_k)$ can communicate with $out(t'_1, ..., t'_m)$ to $com(t_1, ..., t_k)$ provided $k = m$ and t_i and t'_i denote the same data element for $i = 1, ..., k$.

Processes are declared using the keyword **proc**. An example is

$$\begin{array}{ll} \textbf{proc} & counter(x : Nat) = p \\ & buffer = q \end{array}$$

In the first line a counter is declared. It is a process with one parameter x of sort Nat. The parameter x may be used in the process term p that specifies its behaviour. In the second line a parameterless process $buffer$ is declared. Its behaviour is given by the process term q.

Definition 2.1.2 *(Process terms).* An expression p is called a *process term* iff p has the following syntax:

$$\begin{array}{rcl} p & ::= & (p+p) \mid (p \cdot p) \mid (p \parallel p) \mid (p \mathbin{\lfloor\!\lfloor} p) \mid (p \mid p) \mid (p \lhd t \rhd p) \mid \Sigma(d:D,p) \mid \\ & & \partial(\{n_1,...,n_m\},p) \mid \tau(\{n_1,...,n_m\},p) \mid \rho(\{n_1 \to n_1',...,n_m \to n_m'\},p) \mid \\ & & \delta \mid \tau \mid n \mid n(t_1,...,t_m). \end{array}$$

where the n, n_i, n_i' are *names*, the t, t_i stand for data terms, d is a variable and D denotes a sort name. □

Most operators stem from ACP [BW90]. Only the conditional construct $p \lhd t \rhd p$ is taken from [HHJ+87] (see also [BB90]). In process terms we omit brackets according to the convention that $\cdot$ binds strongest, the conditional construct binds stronger than the parallel operators which in turn bind stronger than $+$.

We give a short description of the behaviour represented by closed process terms.

- The $+$ denotes the alternative composition. The process $p+q$ has the same behaviour as the argument that performs the first step.
- The $\cdot$ represents the sequential composition operator. The process $p \cdot q$ behaves as p, and in case p terminates, it continues to behave as q.
- The merge (or parallel composition operator) $\parallel$ denotes the interleaving of its arguments, except that actions from both arguments may communicate if explicitly allowed in a communication specification.
- The left merge $\lfloor\!\lfloor$ and the communication merge $\mid$ are auxiliary operators, to be used for analytical purposes. The left merge is as the merge, except that the first step of $p \mathbin{\lfloor\!\lfloor} q$ must originate from p. The communication merge $\mid$ is also as the merge, except that $p \mid q$ has a communication action between p and q as its first step.
- The *conditional* construct $p \lhd t \rhd q$ is an alternative way to write an **if - then - else**-expression and is introduced by HOARE cs. [HHJ+87]. The data term t is supposed to be of the standard sort of the Booleans (**Bool**). The process $p \lhd t \rhd q$ behaves as p if the data term t evaluates to true (T) and it behaves as q if t evaluates to false (F).
- The sum operator is used to declare a variable d of a specific sort D for use in a process term p. The scope of the variable d is exactly the process term mentioned in the sum operator. The behaviour associated to $\Sigma(d:D,p)$ is a choice between the instantiations of the process term p with values of the sort of the variable d.
- The encapsulation operator (∂) and the hiding operator (τ) are used to rename the action labels $n_1,...,n_m$ to δ, resp. τ. The renaming operator ρ renames action labels according to the scheme in its first argument.
- The constants δ and τ describe two basic types of behaviour. The constant δ describes the process that cannot do anything, in particular it cannot terminate. The constant τ can be used to represent internal activity that cannot be observed.

- The terms n and $n(t_1, ..., t_m)$ represent either process instantiations or actions: n refers to a declared process (or to an action) without parameters and $n(t_1, ..., t_m)$ contains the arguments (i.e. the data terms) of the identifier.

A complete μCRL-*specification* consists of an interleaving of sort, function, axiom, action, communication and process declarations. We provide no modular structuring mechanism. Structuring and organising a specification is up to the specifier. As an example we give a specification of a data transfer process TR. Data elements of sort D are transferred from in to out.

$$
\begin{array}{ll}
\textbf{sort} & \textbf{Bool} \\
\textbf{func} & T, F :\rightarrow \textbf{Bool} \\
\textbf{sort} & D \\
\textbf{func} & d1, d2, d3 :\rightarrow D \\
\textbf{act} & in, out : D \\
\textbf{proc} & TR = \Sigma(d : D, in(x) \cdot out(x) \cdot TR)
\end{array}
$$

2.2 Static, algebraic and operational semantics.

This section explains how the semantics of μCRL is organised. First we shortly describe the 'static' semantics of a specification, i.e. the circumstances under which it is correctly defined. This is the case if all objects that are used are declared exactly once and are used such that the sorts are correct. Furthermore it must be the case that action labels and process names cannot be mixed up and that constant and variable names cannot be confused. Finally, it should be the case that communications are specified in a functional way and that the rewrite rules satisfy the (usual) condition that the variables used at the right-hand side of an equality sign must also occur at the left-hand side. Because all these properties can be statically decided, a specification that is internally consistent is called SSC (*Statically Semantically Correct*).

We say that a μCRL-specification is *well-formed* if it is SSC, it has no empty sorts (which can easily be checked), the communication function is associative and the Booleans are defined. In [GP90] the concepts 'SSC' and 'well-formed' are defined in a precise manner.

For any well-formed specification E its *algebraic* semantics is defined as follows. If Σ is the signature of the data part of E, i.e. all function symbols that are declared in E, then any minimal Σ-algebra that satisfies the axioms in E *and* that contains exactly two elements of sort **Bool** is considered as a model of E. We call this latter property *boolean preserving*, and requiring this property guarantees that the conditional construct behaves as expected.

Based on this algebraic semantics a structured *operational* semantics for processes specified by a well-formed specification E has been defined in the standard way [GP90, GV89]. The idea is that, given some model $\mathbb{A}$ of E, any closed process term yields a labelled transition system of which the labels can be instantiated with (a preferred representation of) closed data terms. The notation $p \leftrightarrow_{\mathbb{A}} q$ then expresses that the transition systems associated with the process terms p and q are *bisimilar* [Par81]. This relation is a congruence w.r.t. the operators of μCRL, and it is the basic equality relation on process terms that we consider. However, we leave it open to consider other (coarser) congruences, provided these are *representation insensitive*, i.e. the equivalence of process terms is invariant under the actual representation of data terms.

3 Syntax and semantics of property formulas

In this section we introduce 'property formulas' with which we can express properties that a specification may have. We provide their syntax and semantics and we introduce variables and substitutions. In the sequel of this paper we adopt the following conventions:

1. We will only consider μCRL-specifications that are well-formed, and further call these simply 'specifications'.
2. Concerning the *names* declared in a specification E: a name n is a *function* from E if E contains a function declaration of the form $n : S_1 \times ... \times S_m \rightarrow S$, where S_i, S are names of sorts declared in E. If $m = 0$ we call n a *constant.* A name n is an *action* from E if it is declared as such, it is a *process* if E contains a process declaration $n = p$.

3.1 Variables and substitutions

In order to express general properties that a specification may have we introduce *variables.* We further introduce *substitutions* to extract the precise instances we are interested in. As properties always refer to a particular specification and as we are dealing with names in a very precise and restrictive way, we define both these concepts relative to the signature of a specification.

Definition 3.1.1 *(Data and process variables).* Let E be a specification. A finite set V_d containing elements of the form $\langle d : D\rangle$ with d some *name* is called a *set of data variables over* E iff

- the name D is declared as a sort in E,
- d is not a constant, or an unparameterised action or process from E,
- for each sort name $D' \not\equiv D$ of E it holds that $\langle d : D'\rangle \notin V_d$.

If we are not interested in the sort of d, we just say that d is 'a variable from V_d'.

Given a set V_d of data variables over E, a finite set V_p of *names* is called a set of *process variables over* E *and* V_d iff non of its elements occur as a variable in V_d. $\Box$

We generally use triples E, V_d, V_p, meaning that E is a (well-formed) specification, V_d is a set of data variables over E, and V_p is a set of process variables over E and V_d. Given E, V_d, V_p, we define many sorted terms that may contain variables. We distinguish two kinds of such terms: data terms and process terms.

Definition 3.1.2 *(Data terms and process terms).* A *data term over* E, V_d, V_p is either a constant from E, a variable from V_d, or an application of a function from E to data terms over E, V_d, V_p of the appropriate sort. A data term is called *closed* iff it does not contain any variables from V_d. Note that for data terms the actual contents of V_d is not relevant.

A *process term over* E, V_d, V_p is defined inductively over the syntax given in definition 2.1.2:

- $p \circ q$ with $\circ \in \{+, \cdot, \|, \mathbin{|\!\!\lfloor}, \mid, \triangleleft t \triangleright\}$ and t a data term over E, V_d, V_p of sort **Bool**, is a *process term over* E, V_d, V_p if both p and q are,

- $\Sigma(d : D, p)$ is a *process term over* E, V_d, V_p if p is a process term over

$$E, (V_d \setminus \{\langle d : n\rangle \mid n \text{ a name}\}) \cup \{\langle d : D\rangle\}, V_p \setminus \{d\},$$

- $C(\{n_1, ..., n_m\}, p)$ with $C \in \{\partial, \tau\}$ is a *process term over* E, V_d, V_p if p is, and the n_i are labels of actions from E,
- $\rho(\{n_1 \to n'_1, ..., n_m \to n'_m\}, p)$ is a *process term over* E, V_d, V_p if p is, and the n_i are labels of actions from E such that if n_i is an action from E then so is n'_i, and if $n_i : S_1 \times ... \times S_k$ is an action declaration in E then so is $n'_i : S_1 \times ... \times S_k$,
- δ and τ are *process terms over* E, V_d, V_p,
- n is a *process term over* E, V_d, V_p if n is an action or a process from E or if $n \in V_p$,
- $n(t_1, ..., t_m)$ is a *process term over* E, V_d, V_p if either E contains an action declaration of the form $n : S_1 \times ... \times S_m$ or a process declaration of the form $n(x_1 : S_1, ..., x_m : S_m) = q$ and any t_i is a data term over E, V_d, V_p of sort S_i.

A process term is called *closed* iff it does not contain any variables from V_d or V_p. □

Let p be a process term over E, V_d, V_p. We say that an occurrence of a name x is *free* in p iff x is a variable from V_d or V_p and this occurrence of x is not in the scope of $\Sigma(x : D, -)$.

Next we introduce 'substitutions'. We distinguish data substitutions and process substitutions. This simplifies the definition of substitutions on process terms containing the sum operator Σ.

Definition 3.1.3. A *data substitution* σ *over* E, V_d, V_p is a mapping from the elements of V_d to the data terms over E, V_d, V_p that preserves sorts. We say that σ is *ground* iff its range only contains closed data terms. Data substitutions are extended to the data terms over E, V_d, V_p in the usual way:

$$\sigma(d) \stackrel{def}{=} \sigma(\langle d : D\rangle) \text{ if } \langle d : D\rangle \in V_d,$$
$$\sigma(c) \stackrel{def}{=} c \text{ if } c \text{ is a constant from } E,$$
$$\sigma(f(t_1, ..., t_m)) \stackrel{def}{=} f(\sigma(t_1), ..., \sigma(t_m)).$$

□

Definition 3.1.4. A *process substitution* σ *over* E, V_d, V_p is a mapping $\sigma : V_p \to \mathcal{P}$, where $\mathcal{P}$ is the set of process terms over E, V_d, V_p. We say that σ is *ground* iff its range only contains closed process terms. Process substitutions are extended to the data terms over E, V_d, V_p by

$$\sigma(t) \stackrel{def}{=} t$$

for any data term t over E, V_d, V_p. □

We also extend both data and process substitutions to process terms. This allows a uniform definition of proof rules. We define this extension simultaneously:

Definition 3.1.5 *(Substitutions on process terms).* Let $\mathcal{P}$ be the set of process terms over E, V_d, V_p and let σ be either a data substitution or a process substitution over E, V_d, V_p. We extend σ to $\mathcal{P}$ as follows (the only non-trivial cases are the sum operator Σ and process variables):

- $\sigma(p \circ q) \stackrel{def}{=} \sigma(p) \circ \sigma(q)$ for $\circ \in \{+, \cdot, \|, \lfloor\!\lfloor, |\}$,
  $\sigma(p \lhd t \rhd q) \stackrel{def}{=} \sigma(p) \lhd \sigma(t) \rhd \sigma(q)$,
  $\sigma(C(nl, p)) \stackrel{def}{=} C(nl, \sigma(p))$ for $C \in \{\partial, \tau, \rho\}$ and nl being the first argument of C,
  $\sigma(\delta) \stackrel{def}{=} \delta$ and $\sigma(\tau) \stackrel{def}{=} \tau$,
  $\sigma(n(t_1, ..., t_m)) \stackrel{def}{=} n(\sigma(t_1), ..., \sigma(t_m))$.

- For a process term $\Sigma(d : D, p) \in \mathcal{P}$ let e be some name not in V_d or V_p such that p is a process term over

  $$E, (V_d \setminus \{\langle d : n\rangle \mid n \text{ a name}\}) \cup \{\langle d : D\rangle, \langle e : D\rangle\}, V_p \setminus \{d\}.$$

  So $p[e/d]$ (notation is explained after this definition) is a term over $E, V_d \cup \{\langle e : D\rangle\}, V_p$. We define

  $$\sigma(\Sigma(d : D, p)) \stackrel{def}{=} \Sigma(e : D, \sigma'(p[e/d]))$$

  where

  - if σ is a *data* substitution, σ' is the data substitution over $E, V_d \cup \{\langle e : D\rangle\}, V_p$ defined by

    $$\sigma'(\langle x : S\rangle) \stackrel{def}{=} \begin{cases} e & \text{if } x \equiv e, \\ \sigma(\langle x : S\rangle) & \text{otherwise,} \end{cases}$$

  - in the case that σ is a *process* substitution, σ' is the process substitution over $E, V_d \cup \{\langle e : D\rangle\}, V_p$ equal to σ.

- For a name $n \in \mathcal{P}$ we define

  $$\sigma(n) \stackrel{def}{=} \begin{cases} \sigma(n) & \text{if } \sigma \text{ is a \textit{process} substitution and } n \in V_p, \\ n & \text{otherwise.} \end{cases}$$

□

If σ is a substitution over E, V_d, V_p that maps variables $x_1, ..., x_m$ to terms $t_1, ..., t_m$, respectively, and that is the identity for any other variable, we use the abbreviation

$$u[t_1, ..., t_m/x_1, ..., x_m] \stackrel{def}{=} \sigma(u)$$

for any term u over E, V_d, V_p. Furthermore, given E, V_d, V_p we sometimes write $p(x_1, ..., x_m)$ for a process term p that possibly contains the data variables $x_1, ..., x_m$ from V_d. In this case we write $p(t_1, ..., t_m)$ for $p[t_1, ..., t_m/x_1, ..., x_m]$, the simultaneous substitution of t_i for x_i.

3.2 Syntax of property formulas

In this section we define 'property formulas' to express properties of specifications. A property formula consists of two parts. The first part is the *property*, which is either an identity between terms, or an application of the operators $\mathcal{F}$ ("Falsum"), $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ known from propositional logic (see eg. [Dal83]) between such identities. The second part of a property formula contains the names of the specification and variable sets.

Definition 3.2.1. A *property over* E, V_d, V_p is defined inductively in the following way:

- $\mathcal{F}$ is a *property over* E, V_d, V_p,
- $t = u$ is a *property over* E, V_d, V_p iff
  - either t and u are data terms over E, V_d, V_p that are of the same sort,
  - or t and u are process terms over E, V_d, V_p,
- $\neg(\phi)$ is a *property over* E, V_d, V_p iff ϕ is a property over E, V_d, V_p,
- $(\phi \circ \psi)$ with $\circ \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$ is a *property over* E, V_d, V_p iff both ϕ and ψ are properties over E, V_d, V_p.

□

Example 3.2.2. Let E be the specification defined in example 2.1.1. Then

$$(times(x, x) = x \ \rightarrow \ (x = 0 \ \vee \ x = S(0)))$$

is a property over $E, \{\langle x : Nat \rangle\}, \emptyset$. *(End example.)*

In properties we omit brackets according to the convention that $=$ binds stronger than any of the logical operators $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$, that $\neg$ binds stronger than any of the logical binary operators, and that $\vee, \wedge$ bind stronger than $\rightarrow, \leftrightarrow$.

For notational convenience we extend the domain of substitutions to properties.

Definition 3.2.3 *(Substitutions on properties).* Let σ be either a data substitution or a process substitution over E, V_d, V_p. We extend σ to the properties over E, V_d, V_p as follows:

$$\begin{array}{lcl}
\sigma(\mathcal{F}) & \stackrel{def}{=} & \mathcal{F}, \\
\sigma(t = u) & \stackrel{def}{=} & \sigma(t) = \sigma(u), \\
\sigma(\neg\phi) & \stackrel{def}{=} & \neg(\sigma(\phi)), \\
\sigma(\phi \circ \psi) & \stackrel{def}{=} & \sigma(\phi) \circ \sigma(\psi) \text{ where } \circ \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}.
\end{array}$$

□

Now a 'property formula' simply consist of a property that has as an attribute the originating specification and variable sets:

Definition 3.2.4. A *property-formula* is an expression of the form

$$\phi \ \mathbf{from}\ E, V_d, V_p$$

where ϕ is a property over E, V_d, V_p. A property formula ϕ **from** E, V_d, V_p is called *closed* iff ϕ contains neither variables from V_d, nor process variables from V_p. □

Note that if ϕ **from** E, V_d, V_p is a property formula and σ is a (process or data) substitution over E, V_d, V_p, then $\sigma(\phi)$ **from** E, V_d, V_p is also a property formula.

We have introduced more logical symbols than strictly necessary for expressing the properties we are interested in. We regard the symbols $\rightarrow$ and $\mathcal{F}$ as basic, and use the other symbols as abbreviations:

Definition 3.2.5. The logical symbols $\neg, \vee, \wedge, \leftrightarrow$ are defined as follows:

$$\begin{aligned}
\neg\phi &\stackrel{def}{=} \phi \rightarrow \mathcal{F},\\
\phi \vee \psi &\stackrel{def}{=} \neg\phi \rightarrow \psi,\\
\phi \wedge \psi &\stackrel{def}{=} \neg(\phi \rightarrow \neg\psi),\\
\phi \leftrightarrow \psi &\stackrel{def}{=} (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi).
\end{aligned}$$

□

3.3 Semantics of property formulas

In this section we define whenever a property-formula ϕ **from** E, V_d, V_p is *valid in* $\mathbb{A}, \approx_{\mathbb{A}}$, notation

$$\mathbb{A}, \approx_{\mathbb{A}} \models \phi \ \mathbf{from}\ E, V_d, V_p$$

(see for $\mathbb{A}$ and $\approx_{\mathbb{A}}$ the definition below). We use the notation

$$\mathbb{A}, \approx_{\mathbb{A}} \not\models \phi \ \mathbf{from}\ E, V_d, V_p$$

if it is not the case that $\mathbb{A}, \approx_{\mathbb{A}} \models \phi$ **from** E, V_d, V_p.

Definition 3.3.1 *(Interpretation of property formulas).* Let E be a specification and $\mathbb{A}$ be a minimal, boolean preserving algebra that is a model of E (see [GP90]). Let furthermore $\approx_{\mathbb{A}}$ be a congruence relation on the closed process terms over E such that $\approx_{\mathbb{A}} \supseteq \,\leftrightarroweq_{\mathbb{A}}$ and such that $\approx_{\mathbb{A}}$ is representation insensitive.

We define the validity of property formulas in two steps:

1. The validity of a *closed* property formula ϕ **from** E, V_d, V_p in $\mathbb{A}, \approx_{\mathbb{A}}$ is defined by induction on the syntax of the property ϕ:

   $\mathbb{A}, \approx_{\mathbb{A}} \not\models \mathcal{F}$ **from** E, V_d, V_p,

   $\mathbb{A}, \approx_{\mathbb{A}} \models t = u$ **from** E, V_d, V_p for data terms t and u iff $\mathbb{A} \models t = u$,

   $\mathbb{A}, \approx_{\mathbb{A}} \models p = q$ **from** E, V_d, V_p for process terms p and q iff $p \approx_{\mathbb{A}} q$,

   $\mathbb{A}, \approx_{\mathbb{A}} \models \phi \rightarrow \psi$ **from** E, V_d, V_p iff

   $$\mathbb{A}, \approx_{\mathbb{A}} \not\models \phi \ \mathbf{from}\ E, V_d, V_p \quad \text{or} \quad \mathbb{A}, \approx_{\mathbb{A}} \models \psi \ \mathbf{from}\ E, V_d, V_p.$$

2. A property formula ϕ **from** E, V_d, V_p is valid in $\mathbb{A}, \approx_{\mathbb{A}}$ iff

$$\mathbb{A}, \approx_{\mathbb{A}} \models \sigma'(\sigma(\phi)) \textbf{ from } E, V_d, V_p$$

for any ground process substitution σ over E, V_d, V_p and any ground data substitution σ' over E, V_d, V_p.

□

Note that in clause 2 of this definition it holds that $\sigma'(\sigma(\phi)) \equiv \sigma(\sigma'(\phi))$ because σ is a ground process substitution.

4 Proof system

We give a proof system in a 'natural deduction' format in which we can derive property formulas. Natural deduction provides rules that agree well with informal reasoning, and is well-known as a formal system of logic. Furthermore the correspondence with proof systems suitable for automatic reasoning is also widely studied [GTL89]. Our set-up is based on [TD88]; other references on natural deduction are eg. [Dal83, Sza69].

Deductions can be constructed according to three types of rules:

1. 'Logical' rules, defining the relations between property formulas that depend on the meaning of the logical symbols, the equality relation and substitution.
2. Rules by which identities between data terms depending on the particular contents of a μCRL specification can be derived.
3. Rules by which identities between process terms depending on the particular contents of a μCRL specification can be derived.

In the next section we introduce the logical rules of our proof system, and present a formal definition of deductions.

4.1 Logical deductions

A deduction can be seen as a tree of which each node is labelled with a property formula (and possibly the name of a rule which has been applied to obtain the property formula). The leaves of the tree are the *assumptions* (also called hypotheses) of the deduction. We use symbols $\mathcal{D}$, possibly subscripted, for arbitrary deductions. We write

$$\begin{array}{c}\mathcal{D}\\ \psi \textbf{ from } E, V_d, V_p\end{array}$$

to indicate that $\mathcal{D}$ has *conclusion*

$$\psi \textbf{ from } E, V_d, V_p$$

(so the occurrence ψ **from** E, V_d, V_p is part of $\mathcal{D}$ itself). We use the notation

$$[\phi \textbf{ from } E, V_d, V_p]$$

for a possibly empty set of occurrences of a property formula ϕ **from** E, V_d, V_p in a deduction, thus

$$\begin{array}{c}[\phi \textbf{ from } E, V_d, V_p]\\ \mathcal{D}\end{array}$$

is a deduction $\mathcal{D}$ with a set $[\phi$ **from** $E, V_d, V_p]$ of assumptions in $\mathcal{D}$. As a rule we assume that

$$[\phi \textbf{ from } E, V_d, V_p]$$

refers to *all* assumptions of the form ϕ **from** E, V_d, V_p in $\mathcal{D}$.

We define logical deductions in a recursive way (recall that $\neg\phi$ abbreviates $\phi \to \mathcal{F}$).

Definition 4.1.1 *(Logical deductions).*

- The single-node tree with as label a property formula ϕ **from** E, V_d, V_p is a deduction from the open assumption ϕ **from** E, V_d, V_p. There are no cancelled assumptions.
- Let $\mathcal{D}_1, \mathcal{D}_2$ be deductions. A new deduction can be constructed according to the rules in table 1. These rules are subject to the following restrictions:
  1. In applications of the introduction rule $\to$I and the rule RAA (Reductio Ad Absurdum) *all* open assumptions of the form indicated by [...] are cancelled.
  2. In applications of $\to$I, RAA, the reflexivity rule REFL, the variable rule VAR and the substitution rule SUB the conclusion should be a property formula.
  3. In applications of SUB the variable x may not be free in any (uncancelled) hypothesis of $\mathcal{D}_1$.
  4. Each application of VAR is restricted to one of the following two cases:
     (a) $V_d \subseteq V'_d$ or $V'_d \subseteq V_d$, and $V_p = V'_p$,
     (b) $V_p \subseteq V'_p$ or $V'_p \subseteq V_p$, and $V_d = V'_d$.

□

The reflexivity rule REFL has an empty premiss, and is therefore called an 'axiom'. The rule VAR is a structural rule that allows (restricted) replacement of variable sets. In the next section we introduce axioms that specify the minimal variable sets involved. With VAR we can obtain variable sets that are suitable for further derivations.

In most deductions the form of the property formulas itself already determines which rule is being applied. Therefore we often omit the names of the rules in deductions. A method that helps to grasp the structure of a given deduction is to number the occurrences of assumptions which are being cancelled, and to repeat the number near the node where the cancellation takes place. Assumptions which are cancelled simultaneously may be given the same number. However, the numbering of discharged assumptions is redundant: by definition any assumption is cancelled at the earliest opportunity. We provide some examples of typical deductions.

$$\frac{\begin{array}{c}[\phi \textbf{ from } E, V_d, V_p]\\ \mathcal{D}_1\\ \psi \textbf{ from } E, V_d, V_p\end{array}}{\phi \to \psi \textbf{ from } E, V_d, V_p} \to\text{I} \qquad \frac{\begin{array}{c}\mathcal{D}_1\\ \phi \textbf{ from } E, V_d, V_p\end{array} \quad \begin{array}{c}\mathcal{D}_2\\ \phi \to \psi \textbf{ from } E, V_d, V_p\end{array}}{\psi \textbf{ from } E, V_d, V_p} \to\text{E}$$

$$\frac{\begin{array}{c}[\neg\phi \textbf{ from } E, V_d, V_p]\\ \mathcal{D}_1\\ \mathcal{F} \textbf{ from } E, V_d, V_p\end{array}}{\phi \textbf{ from } E, V_d, V_p} \text{RAA} \qquad \frac{}{t = t \textbf{ from } E, V_d, V_p} \text{REFL}$$

$$\frac{\begin{array}{c}\mathcal{D}_1\\ \phi[t/x] \textbf{ from } E, V_d, V_p\end{array} \quad \begin{array}{c}\mathcal{D}_2\\ t = u \textbf{ from } E, V_d, V_p\end{array}}{\phi[u/x] \textbf{ from } E, V_d, V_p} \text{REPL}$$

$$\frac{\begin{array}{c}\mathcal{D}_1\\ \phi \textbf{ from } E, V_d, V_p\end{array}}{\phi[t/x] \textbf{ from } E, V_d, V_p} \text{SUB} \qquad \frac{\begin{array}{c}\mathcal{D}_1\\ \phi \textbf{ from } E, V_d, V_p\end{array}}{\phi \textbf{ from } E, V_d', V_p'} \text{VAR}$$

Table 1: Rules for logical deductions

Example 4.1.2. Let ϕ **from** E, V_d, V_p and ψ **from** E, V_d, V_p be two property formulas. We derive

$$\frac{\dfrac{\phi \textbf{ from } E, V_d, V_p \ ^{(1)}}{\psi \to \phi \textbf{ from } E, V_d, V_p} \to\text{I}}{\phi \to (\psi \to \phi) \textbf{ from } E, V_d, V_p} \to\text{I, [1]}$$

Here the [1] in '$\to$I, [1]' indicates that the assumption ϕ **from** E, V_d, V_p $^{(1)}$ is cancelled. *(End example.)*

Example 4.1.3. We here show how to derive the congruence properties of the equality relation $=$ over data terms. Let t, u, v be data terms of sort D, and $t = u$ **from** E, V_d, V_p and $v = t$ **from** E, V_d, V_p be property formulas. Let furthermore x be a name not occurring in V_d or V_p and $V_d' \stackrel{def}{=} V_d \cup \{\langle x : D\rangle\}$.

Reflexivity. Immediate by the axiom REFL.

Symmetry. In the application of the replacement rule REPL we take $\phi \equiv x = t$, so that $\phi[t/x] \equiv t = t$ and $\phi[u/x] \equiv u = t$ (as x occurs not in t or u).

$$\dfrac{\dfrac{\dfrac{}{t = t \textbf{ from } E, V_d', V_p}\,\text{REFL} \qquad \dfrac{t = u \textbf{ from } E, V_d, V_p}{t = u \textbf{ from } E, V_d', V_p}\,\text{VAR}}{u = t \textbf{ from } E, V_d', V_p}\,\text{REPL}}{u = t \textbf{ from } E, V_d, V_p}\,\text{VAR}$$

Transitivity. Take $\phi \equiv v = x$ in the application of REPL with the substitution $[t/x]$:

$$\dfrac{\dfrac{\dfrac{v = t \textbf{ from } E, V_d, V_p}{v = t \textbf{ from } E, V_d', V_p}\,\text{VAR} \qquad \dfrac{t = u \textbf{ from } E, V_d, V_p}{t = u \textbf{ from } E, V_d', V_p}\,\text{VAR}}{v = u \textbf{ from } E, V_d', V_p}\,\text{REPL}}{v = u \textbf{ from } E, V_d, V_p}\,\text{VAR}$$

Substitutivity. Let w be some (process or data) term over E, V_d, V_p and let $[t/z]$, $[u/z]$ be data substitutions over E, V_d, V_p. Take $\phi \equiv w[t/z] = w[x/z]$, and apply REPL with the substitution $[t/x]$:

$$\dfrac{\dfrac{\dfrac{}{w[t/z] = w[t/z] \textbf{ from } E, V_d', V_p}\,\text{REFL} \qquad \dfrac{t = u \textbf{ from } E, V_d, V_p}{t = u \textbf{ from } E, V_d', V_p}\,\text{VAR}}{w[t/z] = w[u/z] \textbf{ from } E, V_d', V_p}\,\text{REPL}}{w[t/z] = w[u/z] \textbf{ from } E, V_d, V_p}\,\text{VAR}$$

In a similar way it can also be proved that $=$ is a congruence relation over process terms. *(End example.)*

Definition 4.1.4 *(Derivability).* Let Γ be a set of property formulas. We write

$$\Gamma \vdash \phi \textbf{ from } E, V_d, V_p$$

iff there is a deduction with all uncancelled assumptions in Γ, and with ϕ **from** E, V_d, V_p as conclusion. In this case we say that there is a *proof* of ϕ **from** E, V_d, V_p from Γ. If $\Gamma = \emptyset$ we just write $\vdash \phi$ **from** E, V_d, V_p and say that ϕ **from** E, V_d, V_p is *logically valid.* □

We state without proof:

Theorem 4.1.5. *(Deduction Theorem.) We have the following standard theorem concerning the derivability of property formulas:*

$$\Gamma \cup \{\phi \textbf{ from } E, V_d, V_p\} \vdash \psi \textbf{ from } E, V_d, V_p \iff \Gamma \vdash \phi \to \psi \textbf{ from } E, V_d, V_p.$$

□

We adopt the following two conventions. If in a derivation only property formulas over fixed E, V_d, V_p are considered, we often leave out the additions ‘ **from** E, V_d, V_p’.

Furthermore, once

$$\{\phi_1 \textbf{ from } E, V_d^1, V_p^1 \ , \ \ldots \ , \ \phi_n \textbf{ from } E, V_d^n, V_p^n\} \vdash \phi \textbf{ from } E, V_d, V_p$$

is proved, the deduction step

$$\frac{\phi_1 \textbf{ from } E, V_d^1, V_p^1 \ , \ \ldots \ , \ \phi_n \textbf{ from } E, V_d^n, V_p^n}{\phi \textbf{ from } E, V_d, V_p}$$

(possibly labelled with some identifier of the proof) may be used in other deductions.
The following lemma provides some standard results.

Lemma 4.1.6. *Let E, V_d, V_p be given. It holds that*

1. $\vdash \phi \to (\psi \to \phi)$,
2. $\{t = u\} \vdash u = t$,
3. $\{v = t,\ t = u\} \vdash v = u$,
4. $\{t = u\} \vdash w[t/z] = w[u/z]$,
5. $\{\phi \to \psi,\ \psi \to \chi\} \vdash \phi \to \chi$,
6. $\{\phi \to \psi\} \vdash \neg\psi \to \neg\phi$,
7. $\{\phi \to \psi,\ \neg\phi \to \psi\} \vdash \psi$

where in 4 it is assumed that w is a (process or data) term over E, V_d, V_p and $[t/z]$, $[u/z]$ are substitutions over E, V_d, V_p.

Proof. Result 1 is proved in example 4.1.2, and 2,3 and 4 are proved in 4.1.3. The results 5 and 6 are standard in propositional logic (derivations can be found in [Dal83]) and we give a proof of 7:

$$\dfrac{\dfrac{\dfrac{\dfrac{\phi \to \psi}{\neg\psi \to \neg\phi}\,(\text{by } 6) \qquad \neg\phi \to \psi}{\neg\psi \to \psi}\,(\text{by } 5) \qquad \neg\psi\ ^{(1)}}{\psi}\,{\to}\mathrm{E} \qquad \neg\psi\ ^{(1)}}{\dfrac{\mathcal{F}}{\psi}\,\text{RAA, [1]}}\,{\to}\mathrm{E}$$

□

Given the abbreviations for the connectives $\vee$ and $\wedge$ in definition 3.2.5, we can *derive* the following deduction rules.

Definition 4.1.7 *(The other connectives).* Let $\mathcal{D}_1, \mathcal{D}_2$ be deductions. A new deduction containing the connectives $\vee$ and $\wedge$ may be constructed according to the rules in table 2. These rules are subject to the following restrictions:

1. In the introduction rules $\vee\mathrm{I}_r$ and and $\vee\mathrm{I}_l$ the conclusion should be a property formula.

2. In the elimination rule $\vee$E all open assumptions ϕ **from** E, V_d, V_p and ψ **from** E, V_d, V_p are cancelled.

□

$$\dfrac{\begin{matrix}\mathcal{D}_1\\ \phi \textbf{ from } E, V_d, V_p\end{matrix}}{\phi \vee \psi \textbf{ from } E, V_d, V_p}\ \vee\mathrm{I}_r \qquad\qquad \dfrac{\begin{matrix}\mathcal{D}_1\\ \phi \textbf{ from } E, V_d, V_p\end{matrix}}{\psi \vee \phi \textbf{ from } E, V_d, V_p}\ \vee\mathrm{I}_l$$

$$\dfrac{\begin{matrix}\mathcal{D}_1\\ \phi \vee \psi \textbf{ from } E, V_d, V_p\end{matrix} \qquad \begin{matrix}[\phi \textbf{ from } E, V_d, V_p]\\ \mathcal{D}_2\\ \chi \textbf{ from } E, V_d, V_p\end{matrix} \qquad \begin{matrix}[\psi \textbf{ from } E, V_d, V_p]\\ \mathcal{D}_3\\ \chi \textbf{ from } E, V_d, V_p\end{matrix}}{\chi \textbf{ from } E, V_d, V_p}\ \vee\mathrm{E}$$

$$\dfrac{\begin{matrix}\mathcal{D}_1\\ \phi \textbf{ from } E, V_d, V_p\end{matrix} \qquad \begin{matrix}\mathcal{D}_2\\ \psi \textbf{ from } E, V_d, V_p\end{matrix}}{\phi \wedge \psi \textbf{ from } E, V_d, V_p}\ \wedge\mathrm{I}$$

$$\dfrac{\begin{matrix}\mathcal{D}_1\\ \phi \wedge \psi \textbf{ from } E, V_d, V_p\end{matrix}}{\phi \textbf{ from } E, V_d, V_p}\ \wedge\mathrm{E}_r \qquad\qquad \dfrac{\begin{matrix}\mathcal{D}_1\\ \phi \wedge \psi \textbf{ from } E, V_d, V_p\end{matrix}}{\psi \textbf{ from } E, V_d, V_p}\ \wedge\mathrm{E}_l$$

Table 2: Rules for the other logical connectives

Whenever convenient, we prove results with the help of these derivable rules. As an example we show the derivability of the rule $\vee$E, where the double bar indicates the abbreviation of $\vee$:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\begin{matrix}\phi^{\ (1)}\\ \mathcal{D}_2\\ \chi\end{matrix} \qquad \neg\chi^{\ (2)}}{\mathcal{F}}\ {\to}\mathrm{E}}{\neg\phi}\ \text{RAA, [1]} \qquad \cfrac{\begin{matrix}\mathcal{D}_1\\ \phi\vee\psi\end{matrix}}{\neg\phi\to\psi}}{\psi}\ {\to}\mathrm{E} \qquad \cfrac{\cfrac{\begin{matrix}\psi^{\ (3)}\\ \mathcal{D}_3\\ \chi\end{matrix} \qquad \neg\chi^{\ (2)}}{\mathcal{F}}\ {\to}\mathrm{E}}{\neg\psi}\ \text{RAA, [3]}}{\mathcal{F}}\ {\to}\mathrm{E}}{\chi}\ \text{RAA, [2]}$$

For readability we further introduce the notations

$$\bigvee_{i\in I}\phi_i \textbf{ from } E, V_d, V_p \quad \text{and} \quad \bigwedge_{i\in I}\phi_i \textbf{ from } E, V_d, V_p$$

for iterated finite disjunctions and conjunctions, respectively. We adopt the convention that

$$\bigvee_{i\in\emptyset}\phi_i \textbf{ from } E, V_d, V_p \stackrel{\mathit{def}}{=} \mathcal{F} \textbf{ from } E, V_d, V_p$$

and

$$\bigwedge_{i\in\emptyset}\phi_i \textbf{ from } E, V_d, V_p \stackrel{\mathit{def}}{=} \neg\mathcal{F} \textbf{ from } E, V_d, V_p.$$

4.2 Modules for data equivalence

As μCRL is based on ACP [BW90] we follow its methodology and consider 'building blocks' of axioms and rules that describe a feature of concurrency in a certain semantical setting. We call such building blocks *modules.* If $M_1, ..., M_n$ are modules, then the notation

$$M_1 + ... + M_n + \Gamma \vdash \phi \textbf{ from } E, V_d, V_p$$

expresses that with the axioms and rules from $M_1, ..., M_n$ we can derive ϕ **from** E, V_d, V_p with all uncancelled assumptions in the set Γ of property formulas.

In this section we introduce three modules that permit us to derive identities between data terms that depend on the contents of a specification.

The module BOOL. Concerning the standard sort **Bool** we define two axioms, corresponding with the demand that any model of a specification E is *boolean preserving*:

$$\frac{}{\neg(T=F) \textbf{ from } E, \emptyset, \emptyset}\ \text{B1}$$

which states that the Booleans T and F are considered different in our proof system, and the axiom

$$\frac{}{\neg(b=T)\to b=F \textbf{ from } E, \{\langle b:\textbf{Bool}\rangle\}, \emptyset}\ \text{B2}$$

which expresses that there are at most two Boolean values, represented by T and F. The two axioms B1 and B2 form the module BOOL. The following lemma states that the reverse implication in B2 is derivable.

Lemma 4.2.1. *For any specification E it holds that*

$$\text{BOOL} \vdash b = F \to \neg(b = T) \textbf{ from } E, \{\langle b : \textbf{Bool} \rangle\}, \emptyset.$$

Proof. In the following deduction, which proves the lemma, we again leave out all the additions **from** ... and only display the properties. However, note that we need an application of the rule VAR that changes the variable set $\emptyset$ from the axiom B1 to the variable set $\{\langle b : \textbf{Bool} \rangle\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{b = F^{\ (1)} \qquad b = T^{\ (2)}}{T = F}\,\text{REPL} \qquad\qquad \dfrac{}{\neg(T = F)}\,\text{B1, VAR}}{\mathcal{F}}\,\to\text{E.}}{\neg(b = T)}\,\text{RAA, [2]}}{b = F \to \neg(b = T)}\,\to\text{I, [1]}$$

□

The module FACT**.** The basic identities on data terms are those declared in a specification E. Assume $t = u$ occurs as an axiom in E, i.e. $t = u$ is preceded by the keyword **rew**. Then we have an axiom

$$\dfrac{}{t = u \textbf{ from } E, V_d, \emptyset}\ \text{FACT}$$

where V_d is the set of data variables occurring in t and u. Note that the module consisting of all the FACTs from E is implicitly present in the E occurring in property formulas. Therefore we generally do not mention FACT before the turnstyle, although it may have been used.

The module IND($\overline{C}$)**.** It is required that any model for the data part is minimal. In the proof theory this can be captured via induction. Therefore we introduce an induction rule. In example 4.2.3, based on example 2.1.1, we illustrate this rule by deriving the commutativity of addition on natural numbers. We start with a preparatory definition.

Definition 4.2.2 *(Constructors).* Let E be a specification, S the name of a sort occurring in E, and C a subset of the function declarations occurring in E. We say that C is a *constructor set of the sort* S iff all functions in C have target sort S, and any closed data term of sort S can be proved equal to a data term that is obtained from applications of the functions in C and terms not of sort S only. □

In general it is not possible to prove that a given set is a constructor set within our framework. Reasons for this are that we can neither express 'existential' properties of data terms, nor that a term is obtained from application of a constructor function. Therefore such a proof must be given on a meta-level. In example 4.2.3, we can prove that $0 :\to Nat$ and $S : Nat \to Nat$ form a constructor set of the sort Nat by using the axioms given there and structural induction on the complexity of closed terms.

Assume that for given E, V_d, V_p we have that

$$\{\langle x_1 : S_1\rangle, .., \langle x_m : S_m\rangle\} \subseteq V_d.$$

Let for $1 \leq i \leq m$:

$$C_i \stackrel{def}{=} \{f_{ij} : S_1^{ij} \times ... \times S_{l_{ij}}^{ij} \to S_i \mid 1 \leq j \leq k_i \ , \ k_i > 0 \ , \ l_{ij} \geq 0\}$$

be a constructor set of the sort S_i of cardinality k_i. We introduce the following induction rule $\text{IND}(C_1, ..., C_m)$ that is parameterised by the constructor sets $C_1, ..., C_m$. The induction takes place on the variables $x_1, ..., x_m$.

$$\frac{\begin{array}{c}\mathcal{D}_{ij} \\ \bigwedge_{\sigma \in I_{ij}} \sigma(\phi) \to \phi[f_{ij}(z_1^{ij}, ..., z_{l_{ij}}^{ij})/x_i] \textbf{ from } E, V_d \cup \{\langle z_n^{ij} : S_n^{ij}\rangle \mid 1 \leq n \leq l_{ij}\}, V_p\end{array}}{\phi \textbf{ from } E, V_d, V_p} \begin{array}{l} {\scriptstyle 1 \leq i \leq m} \\ {\scriptstyle 0 \leq j \leq k_i} \end{array}$$

where for each $1 \leq i \leq m$ and $1 \leq j \leq k_i$ the index set I_{ij} is a set of data substitutions over $E, V_d \cup \{\langle z_n^{ij} : S_n^{ij}\rangle \mid 1 \leq n \leq l_{ij}\}, V_p$ satisfying for $1 \leq k \leq m$:

$\sigma \in I_{ij} \iff \sigma$ is the identity, except that it maps x_k to some y_k, where

- $y_k \in \{x_1, ..., x_m\} \ \cup \ \{z_n^{ij} \mid 1 \leq n \leq l_{ij}\}$,
- $y_i \not\equiv x_i$,
- if $1 \leq k < k' \leq m$, then $y_k \not\equiv y_{k'}$.

Note that in $\text{IND}(C_1, ..., C_m)$ all the variables $x_1, ..., x_m, z_1^{ij}, ..., z_{l_{ij}}^{ij}$ are pairwise different for all appropriate i, j. In section 4.4 we give an argument for its soundness.

Example 4.2.3. Let E be the specification from example 2.1.1:

sort Nat
func $0 :\to Nat$
$S : Nat \to Nat$
$add, times : Nat \times Nat \to Nat$
var $x, y : Nat$
rew $add(x, 0) = x$
$add(x, S(y)) = S(add(x, y))$
$times(x, 0) = 0$
$times(x, S(y)) = add(x, times(x, y))$

We prove that the function add is commutative, i.e. $add(x, y) = add(y, x)$ **from** $E, V_d, \emptyset$, where $V_d \stackrel{def}{=} \{\langle x : Nat\rangle, \langle y : Nat\rangle, \langle z : Nat\rangle\}$. The proof is in four steps:

a. $add(0, x) = x$,
b. $add(S(0), x) = S(x)$,
c. $add(x, add(y, z)) = add(add(x, y), z)$,
d. $add(x, y) = add(y, x)$.

As observed above, we can take $C_1 \stackrel{def}{=} \{0 :\rightarrow Nat, S : Nat \rightarrow Nat\}$ as a set of constructors of sort Nat. Let $f_{10} = 0$ and $f_{11} = S$. We prove a and the final result d, and leave proofs of b and c to the reader. In the following deductions the bars labelled with a $(*)$ refer to lemma 4.1.6.3+4.

Ad a**.** Let $\psi \equiv add(0, x) = x$, then $I_{10} = \emptyset$ and $I_{11} = \{[n/x]\}$ with n a fresh variable of sort Nat.

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{}{add(x,0) = x}\\ \psi[0/x]\end{array}}{\bigwedge_{\sigma \in I_{10}} \sigma(\psi) \rightarrow \psi[0/x]} \qquad \dfrac{\dfrac{\dfrac{}{add(0, S(n)) = S(add(0, n))} \quad \dfrac{\psi[n/x]^{\;(1)}}{S(add(0, n)) = S(n)}\,(*)}{\psi[S(n)/x]}\,(*)}{\bigwedge_{\sigma \in I_{11}} \sigma(\psi) \rightarrow \psi[S(n)/x]}\,[1]}{\psi}\,\text{IND}$$

Ad d**.** Take $\phi \equiv add(x, y) = add(y, x)$. The induction takes place on the variable y, so $I_{10} = \emptyset$ and $I_{11} = \{[n/y]\}$. In the following deduction $\mathcal{D}_1$ abbreviates an easy deduction based on result a, and $\mathcal{D}_2$ abbreviates a simple deduction that uses the second axiom of E, the results b and c, and the congruence properties of $=$ proved in lemma 4.1.6.

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{}{add(0,x) = x}\,(a)\\ \mathcal{D}_1\\ \phi[0/y]\end{array}}{\bigwedge_{\sigma \in I_{10}} \sigma(\phi) \rightarrow \phi[0/y]} \qquad \dfrac{\begin{array}{c}\dfrac{\dfrac{}{add(x, S(n)) = S(add(x, n))} \quad \dfrac{\phi[n/y]^{\;(1)}}{S(add(x, n)) = S(add(n, x))}\,(*)}{add(x, S(n)) = S(add(n, x))}\,(*)\\ \mathcal{D}_2\\ \phi[S(n)/y]\end{array}}{\bigwedge_{\sigma \in I_{11}} \sigma(\phi) \rightarrow \phi[S(n)/y]}\,[1]}{\phi}\,\text{IND}$$

(End example.)

4.3 Modules for process equivalence

In this section we introduce the means to derive identities between process terms using the originating specification and standard process algebra axioms and rules.

The module REC**.** Let for some given E it be the case that $n = p$ is a process declaration in E (i.e. the last keyword preceding $n = p$ is **proc**). Then we have an axiom

$$\dfrac{}{n = p \textbf{ from } E, \emptyset, \emptyset}\ \text{REC}$$

If $n(x_1 : S_1, ..., x_k : S_k) = p$ is a process declaration in E, then we have an axiom

$$\dfrac{}{n(x_1, ..., x_k) = p \textbf{ from } E, \{\langle x_1 : S_1\rangle, ..., \langle x_k : S_k\rangle\}, \emptyset}\ \text{REC}$$

Like in the case of FACT we adopt the convention not to denote the module REC before the turnstyle.

A1	$x + y = y + x$	CF1	$n_1 \mid n_2 = n_3$ if $n_1 \mid n_2 = n_3 \in Comm(E)$
A2	$x + (y + z) = (x + y) + z$		
A3	$x + x = x$	CF1′	$n_1(t_1, ..., t_m) \mid n_2(t_1, ..., t_m) = n_3(t_1, ..., t_m)$
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$		if $n_1 \mid n_2 = n_3 \in Comm(E)$
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$		
A6	$x + \delta = x$	CF2	$a \mid b = \delta$
A7	$\delta \cdot x = \delta$		if $\forall n \in \mathcal{N}$. $label(a) \mid label(b) = n \notin Comm(E)$
		CF2′	$\neg(t_i = t'_i) \rightarrow n_1(t_1, ..., t_m) \mid n_2(t'_1, ..., t'_m) = \delta$
CM1	$x \parallel y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x + x \mid y$		for some $1 \leq i \leq m$
CM2	$a \mathbin{\lfloor\!\lfloor} x = a \cdot x$		
CM3	$a \cdot x \mathbin{\lfloor\!\lfloor} y = a \cdot (x \parallel y)$	CF2″	$n_1(t_1, ..., t_m) \mid n_2(t'_1, ..., t'_{m'}) = \delta$ if $m \neq m'$
CM4	$(x + y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z$		
CM5	$a \cdot x \mid b = (a \mid b) \cdot x$		
CM6	$a \mid b \cdot x = (a \mid b) \cdot x$	D1	$\partial(\{n_1, ..., n_m\}, a) = a$ if $label(a) \notin \{n_1, ..., n_m\}$
CM7	$a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \parallel y)$	D2	$\partial(\{n_1, ..., n_m\}, a) = \delta$ if $label(a) \in \{n_1, ..., n_m\}$
CM8	$(x + y) \mid z = x \mid z + y \mid z$	D3	$\partial(nl, x + y) = \partial(nl, x) + \partial(nl, y)$
CM9	$x \mid (y + z) = x \mid y + x \mid z$	D4	$\partial(nl, x \cdot y) = \partial(nl, x) \cdot \partial(nl, y)$

Table 3: The axioms of ACP for a specification E, where a and b range over δ, τ and the actions of E, the n_i range over $\mathcal{N}$ and $m, m' \geq 1$.

The modules ACP, SC, HIDE **and** REN**.** In table 3 we present the system ACP, consisting of all process algebra axioms that are standard in that theory [BW90]. The axioms CF refer to any specification E, where the set $Comm(E)$ is the commutative and associative closure of all communications declared in E (the well-formedness of E implies that $Comm(E)$ is finite). In CF2, D1 and D2 we use a function $label()$ that extracts the label of an atomic action, and is the identity for δ and τ.

We present in table 4 some axioms for the merge operators, known as the Standard Concurrency laws (see [BW90]). These axioms are derivable for process terms that are constructed from atomic actions, δ and τ.

For hiding (abstraction) we present the module HIDE in table 5, and for general renaming we have the module REN in table 6 available. In both modules the function $label()$ is used again.

Let E be a specification. For any equation ϕ from ACP, SC, HIDE and REN (possibly depending on E) we have an axiom

$$\frac{}{\phi \textbf{ from } E, \emptyset, V_p} \; \textit{name of } \phi$$

where V_p is the set of variables occurring in ϕ.

SC1 $(x \mathbin{\|\!\lfloor} y) \mathbin{\|\!\lfloor} z = x \mathbin{\|\!\lfloor} (y \parallel z)$	SC4 $(x \mid y) \mid z = x \mid (y \mid z)$
SC2 $x \mathbin{\|\!\lfloor} \delta = x$	SC5 $x \mid (y \mathbin{\|\!\lfloor} z) = (x \mid y) \mathbin{\|\!\lfloor} z$
SC3 $x \mid y = y \mid x$	

Table 4: The axioms of SC.

TI1	$\tau(\{n_1, ..., n_m\}, a) = a$	if $label(a) \notin \{n_1, ..., n_m\}$
TI2	$\tau(\{n_1, ..., n_m\}, a) = \tau$	if $label(a) \in \{n_1, ..., n_m\}$
TI3	$\tau(nl, x + y) = \tau(nl, x) + \tau(nl, y)$	
TI4	$\tau(nl, x \cdot y) = \tau(nl, x) \cdot \tau(nl, y)$	

Table 5: The axioms of HIDE for a specification E, where a ranges over δ, τ and the actions of E, the n_i range over $\mathcal{N}$ and $m \geq 1$.

RN1	$\rho(\{n_1 \to n'_1, ..., n_m \to n'_m\}, a) = a$	if $label(a) \notin \{n_1, ..., n_m\}$
RN2	$\rho(\{n_1 \to n'_1, ..., n_m \to n'_m\}, n_i) = n'_i$	if $1 \leq i \leq m$
RN2$'$	$\rho(\{n_1 \to n'_1, ..., n_m \to n'_m\}, n_i(t_1, ..., t_{m'})) = n'_i(t_1, ..., t_{m'})$	if $1 \leq i \leq m$
RN3	$\rho(nl, x + y) = \rho(nl, x) + \rho(nl, y)$	
RN4	$\rho(nl, x \cdot y) = \rho(nl, x) \cdot \rho(nl, y)$	

Table 6: The axioms of REN for a specification E, where a ranges over δ, τ and the actions of E, n_i, n'_i range over $\mathcal{N}$, and $m, m' \geq 1$.

The module COND. We define for any specification E two axioms characterising the behaviour of the conditional [BB90, HHJ+87]:

$$\frac{}{x \lhd T \rhd y = x \textbf{ from } E, \emptyset, \{x, y\}}\ \text{Cond1}$$

and

$$\frac{}{x \lhd F \rhd y = y \textbf{ from } E, \emptyset, \{x, y\}}\ \text{Cond2.}$$

These two axioms form the module COND. The following lemma describes two basic properties that can be proved using the module COND. Both these results will be used later in the paper.

Lemma 4.3.1. *Let E, V_d, V_p be such that $\langle b : \textbf{Bool}\rangle \in V_d$ and $\{x, y, z\} \subseteq V_p$. Then*

1. $\text{BOOL} + \text{ACP} + \text{COND} \vdash x + x \lhd b \rhd \delta = x \textbf{ from } E, V_d, V_p$,
2. $\text{BOOL} + \text{COND} \vdash (b = T \to x = y) \to (x \lhd b \rhd z = y \lhd b \rhd z) \textbf{ from } E, V_d, V_p$.

Proof. In the following deductions the bars labelled with $(*)$ refer to lemma 4.1.6. As we can derive from the axiom B2, i.e.

$$\neg(b = T) \to b = F \textbf{ from } E, \{\langle b : \textbf{Bool}\rangle\}, \emptyset,$$

the property formula

$$T = b \vee F = b \textbf{ from } E, V_d, V_p$$

we can apply the rule $\vee E$ to obtain 1:

$$\frac{\dfrac{}{T = b \vee F = b}\,\text{B2} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \lhd T \rhd y = x}}{x \lhd T \rhd \delta = x} \quad T = b\ ^{(1)}}{x \lhd b \rhd \delta = x}}{x + x \lhd b \rhd \delta = x + x}\,(*) \quad \dfrac{}{x + x = x}\,\text{A3}}{x + x \lhd b \rhd \delta = x}\,(*) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \lhd F \rhd y = y}}{x \lhd F \rhd \delta = \delta} \quad F = b\ ^{(2)}}{x \lhd b \rhd \delta = \delta}}{x + x \lhd b \rhd \delta = x + \delta}\,(*) \quad \dfrac{}{x + \delta = x}\,\text{A6}}{x + x \lhd b \rhd \delta = x}\,(*)}{x + x \lhd b \rhd \delta = x}\ \vee\text{E}, [1, 2]$$

The following deduction proves 2, where $\mathcal{D}$ abbreviates an easy deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{}\;\; b = T \to x = y\ ^{(1)} \qquad \dfrac{\dfrac{x = y\ ^{(2)}}{x \lhd b \rhd z = y \lhd b \rhd z}\,(*)}{x = y \to x \lhd b \rhd z = y \lhd b \rhd z}\,[2]}{b = T \to x \lhd b \rhd z = y \lhd b \rhd z}\,(*) \qquad \dfrac{\begin{array}{c} \dfrac{\neg(b = T)\ ^{(3)} \quad \dfrac{}{\neg(b = T) \to b = F}}{b = F} \\ \mathcal{D} \\ x \lhd b \rhd z = y \lhd b \rhd z \end{array}}{\neg(b = T) \to x \lhd b \rhd z = y \lhd b \rhd z}\,[3]}{x \lhd b \rhd z = y \lhd b \rhd z}\,(*)}{(b = T \to x = y) \to (x \lhd b \rhd z = y \lhd b \rhd z)}\,[1]}{}$$

□

SUM1	$\Sigma(d:D,x)=x$	
SUM2	$\Sigma(d:D,\sigma(x))=\Sigma(e:D,\sigma(x)[e/d])$	provided e not free in $\sigma(x)$
SUM3	$\Sigma(d:D,\sigma(x))=\Sigma(d:D,\sigma(x))+\sigma(x)$	
SUM4	$\Sigma(d:D,\sigma(x)+\sigma(y))=\Sigma(d:D,\sigma(x))+\Sigma(d:D,\sigma(y))$	
SUM5	$\Sigma(d:D,\sigma(x)\cdot y)=\Sigma(d:D,\sigma(x))\cdot y$	
SUM6	$\Sigma(d:D,\sigma(x)\,\lfloor\!\lfloor\, y)=\Sigma(d:D,\sigma(x))\,\lfloor\!\lfloor\, y$	
SUM7	$\Sigma(d:D,\sigma(x)\,\vert\, y)=\Sigma(d:D,\sigma(x))\,\vert\, y$	
SUM8	$\Sigma(d:D,\partial(nl,\sigma(x)))=\partial(nl,\Sigma(d:D,\sigma(x)))$	
SUM9	$\Sigma(d:D,\tau(nl,\sigma(x)))=\tau(nl,\Sigma(d:D,\sigma(x)))$	
SUM10	$\Sigma(d:D,\rho(nl,\sigma(x)))=\rho(nl,\Sigma(d:D,\sigma(x)))$	
SUM11	$\dfrac{\begin{array}{c}\mathcal{D}\\ \sigma(x)=\sigma(y)\ \textbf{from}\ E,V_d,V_p\end{array}}{\Sigma(d:D,\sigma(x))=\Sigma(d:D,\sigma(y))\ \textbf{from}\ E,V_d,V_p}$	provided d not free in the assumptions of $\mathcal{D}$

Table 7: The axioms and congruence rule of SUM for E,V_d,V_p, where E contains a sort name D, $\langle d:D\rangle\in V_d$, V_p contains x, and for SUM4–7 and SUM11 also y, and σ is a process substitution over E,V_d,V_p.

The module SUM. For the sum operator we present the module SUM in table 7. Recall that substitutions are defined in such a way that they never introduce new bindings of variables. In order to describe the general properties of the sum operator, the axioms of SUM are formulated using process substitutions *within* the scope of the Σ (in fact the process terms $\sigma(x)$ and $\sigma(y)$ are used as syntactic variables for process terms). Another consequence of the way we defined substitutions is that the congruence property for the sum operator does not follow from the general replacement rule REPL. This property is separately captured by the rule SUM11 (in the special case that d occurs not free in $\sigma(x)$ and $\sigma(y)$, SUM11 can be derived with REPL and SUM2). For any of the equations ϕ in the module SUM we have an axiom

$$\frac{}{\phi\ \textbf{from}\ E,V_d,V_p}\ \textit{name of}\ \phi$$

where V_d and V_p are chosen minimal.

The sum operator typically describes the alternative composition of all data instances of a process term. This is expressed in the following lemma.

Lemma 4.3.2. *Let E,V_d,V_p be such that the sort D and an equality function eq over D occur in E. Let furthermore $\{\langle d:D\rangle,\langle e:D\rangle\}\subseteq V_d$, and $\mathcal{M}\supseteq\{\mathrm{BOOL},\mathrm{COND},\mathrm{SUM}\}$ be such that $\mathcal{M}\vdash eq(d,e)=T\rightarrow d=e$ **from** E,V_d,V_p. Then for any process term $p(d)$ over E,V_d,V_p it holds that*

$$\mathcal{M}\vdash\Sigma(d:D,p(d))=\Sigma(d:D,\delta\lhd eq(d,e)\rhd p(d))+p(e)\ \textbf{from}\ E,V_d,V_p.$$

Proof. First note that it is very plausible that an 'equality function' eq satisfies the property $eq(d,e) = T \to d = e$. The proof uses the straightforward identity

$$\delta \lhd eq(d,e) \rhd p(d) + p(e) = p(d) + p(e) \textbf{ from } E, V_d, V_p, \tag{1}$$

of which we leave the derivation (in which the property of the function eq is necessary) to the reader. We derive the identity that proves the lemma in a 'linear style' (more on this style in section 5):

$$\begin{aligned}\Sigma(d:D,p(d)) &\overset{\text{SUM3}}{=} \Sigma(d:D,p(d)) + p(e)\\ &\overset{\text{SUM1}}{=} \Sigma(d:D,p(d)) + \Sigma(d:D,p(e))\\ &\overset{\text{SUM4}}{=} \Sigma(d:D,p(d)+p(e))\end{aligned}$$

Hence, using SUM11 and (1) it follows that

$$\begin{aligned}\Sigma(d:D,p(d)) &= \Sigma(d:D,\delta \lhd eq(d,e) \rhd p(d) + p(e))\\ &\overset{\text{SUM4}}{=} \Sigma(d:D,\delta \lhd eq(d,e) \rhd p(d)) + \Sigma(d:D,p(e))\\ &\overset{\text{SUM1}}{=} \Sigma(d:D,\delta \lhd eq(d,e) \rhd p(d)) + p(e)\end{aligned}$$

□

If in lemma 4.3.2 the sort D is *finitely representable*, i.e. there are closed data terms $t_1, ..., t_n$ of sort D such that

$$\bigvee_{i=1}^{n} d = t_i \textbf{ from } E, V_d, V_p$$

is derivable, then it follows that

$$\Sigma(d:D,p(d)) = p(t_1) + ... + p(t_n) \textbf{ from } E, V_d, V_p$$

is also derivable.

The module RSP. In order to derive identities between infinite processes we introduce (an extended version of) the Recursive Specification Principle (RSP, see eg. [BW90]).

The idea of RSP is that if two (different) process terms both satisfy some 'process-equation', then those process terms are considered equal. In general we use a system of such equations, each of which must contain at its left-hand side a (possibly parameterised) fresh identifier and at its right-hand side a 'process term' that may contain the new identifiers. These identifiers may be parameterised with data. We introduce a mechanism that defines substitution of parameterised process terms in a system of process-equations. The soundness of RSP depends on the guardedness of the system of process-equations used. In the following we make all these notions precise, and introduce the rule RSP.

Let E, V_d, V_p be given and let $n_1, ..., n_m$ be m different names. We call a system G of m equations $G_1, ..., G_m$ a system of *process-equations over* E, V_d, V_p iff

1. Each equation G_i has at its left-hand side an expression of the form

$$n_i(x_{i1}, ..., x_{im_i}) \tag{2}$$

where any x_{ij} is a data variable from V_d, or of the form n_i.

2. Let G'_i be as G_i, except that any left-hand side of the form (2) is replaced by $n_i(x_{i1} : S_{i1}, ..., x_{im_i} : S_{im_i})$ where S_{ij} is the sort of x_{ij}. Then the following extension of E must be a well-formed specification.

$$\begin{array}{ll} E & \\ \textbf{proc} & G'_1 \\ & \vdots \\ & G'_m \end{array}$$

This guarantees that any right-hand side of G_i is a proper process term over this extension of E that possibly contains data variables from $\{\langle x_{ij} : S_{ij}\rangle \mid 1 \leq j \leq m_i\}$ (setting $m_i = 0$ in case G_i is not of the form (2)).

Next we introduce a substitution mechanism for a system $G = G_1, ..., G_m$ of process-equations over E, V_d, V_p. Abbreviating the (possible) variables of n_i by $\bar{x}_i$ and writing $<>$ for the empty sequence of variables, we define

$$G_i[\lambda \bar{x}_i \,.\, p(\bar{x}_i)/n_i]$$

as the equation obtained by substituting $\lambda \bar{x}_i \,.\, p(\bar{x}_i)$ for the n_i-occurrences in G_i, and then repeatedly performing β-conversion on the respective arguments of the identifier n_i. For any identifier without arguments only the substitution of p is performed. In example 4.3.4 this substitution mechanism is illustrated.

The rule RSP is restricted to (syntactically) guarded systems of process-equations:

Definition 4.3.3 *(Guardedness of G).* Let G be a system of process-equations over E, V_d, V_p and let N be the left-hand side of one of the equations of G. We say that N is *guarded* in r, where r is a subterm of one of the right-hand sides of G, iff

- $r \equiv q_1 \circ q_2$ with $\circ \in \{+, \|, \mid, \triangleleft t \triangleright\}$, and N is guarded in q_1 and q_2,
- $r \equiv q_1 \circ q_2$ with $\circ \in \{\cdot, \lfloor\!\lfloor\}$ and N is guarded in q_1,
- $r \equiv \Sigma(x : S, q_1)$ and N is guarded in q_1,
- $r \equiv C(nl, q_1)$ with $C \in \{\partial, \tau, \rho\}$ and nl being a list of names (or in the case of ρ a renaming scheme), and N is guarded in q_1,
- $r \equiv \delta$ or $r \equiv \tau$,
- $r \equiv n'$ for a *name* n' and $N \not\equiv n'$,
- $r \equiv n'(u_1, ..., u_{m'})$ and $N \not\equiv n'(x_{i1}, ..., x_{im_i})$.

If N is not guarded in r we say that N appears *unguarded* in r.

The *Identifier Dependency Graph* of G, notation $IDG(G)$, is constructed as follows:

- each left-hand side of the equations of G is a node,
- if N is a node of $IDG(G)$ and $N = r \in G$, then there is an edge $N \to N'$ for any node N' that appears unguarded in r.

We call G *guarded* iff $IDG(G)$ is well founded, i.e. does not contain an infinite path. $\square$

Given a guarded system $G_1, ..., G_m$ of m process-equations over E, V_d, V_p, we define the following rule RSP:

$$\frac{\begin{array}{cc}\mathcal{D}_{1i} & \mathcal{D}_{2i} \\ G_i[\lambda\bar{x}_j\ .\ p_j(\bar{x}_j)/n_j]_{j=1}^m \textbf{ from } E, V_d, V_p & G_i[\lambda\bar{x}_j\ .\ q_j(\bar{x}_j)/n_j]_{j=1}^m \textbf{ from } E, V_d, V_p\end{array}}{p_k(\bar{x}_k) = q_k(\bar{x}_k) \textbf{ from } E, V_d, V_p \quad (1 \leq k \leq m)}\ 1 \leq i \leq m$$

where

- for $1 \leq i \leq m$ the $p_i(\bar{x}_i)$ and $q_i(\bar{x}_i)$ are process terms over E, V_d, V_p,
- the notation $[...]_{j=1}^m$ abbreviates the m given, consecutive substitutions.

In the next section we argue why G has to be guarded. We now give a typical example of an application of RSP.

Example 4.3.4. Consider the following guarded μCRL-specification:

$$\begin{array}{lllll} E \equiv & \textbf{sort} & \textbf{Bool} & & \\ & \textbf{func} & T, F :\to \textbf{Bool} & & \\ & \textbf{sort} & S & & \\ & \textbf{func} & C :\to S & & \\ & & f, g : S \to S & & \\ & \textbf{act} & a & & \\ & \textbf{proc} & p(x:S) & = & a \cdot p(f(x)) + a \\ & & q(x:S) & = & a \cdot q(g(x)) + a \end{array}$$

We want to prove

$$\text{RSP} \vdash p(x) = q(y) \textbf{ from } E, \{\langle x : S\rangle, \langle y : S\rangle\}, \emptyset.$$

Therefore we define a system G as follows:

$$G \stackrel{def}{=} n(x,y) \quad = \quad a \cdot n(f(x), g(y)) + a$$

so that G is guarded. To illustrate the substitution mechanism we first perform the substitution $G[\lambda x, y\ .\ p(x)/n]$ step by step:

1. Substitution $[\lambda x, y\ .\ p(x)/n]$ (underlined) and denoting arguments in β-conversion format (doubly underlined):

$$\underline{\lambda x, y\ .\ p(x)}\ \underline{\underline{x}}\ \underline{\underline{y}} = a \cdot\ \underline{\lambda x, y\ .\ p(x)}\ \underline{\underline{f(x)}}\ \underline{\underline{g(y)}}\ + a.$$

2. Apply β-conversion two times:

$$p(x) = a \cdot p(f(x)) + a.$$

The reader may check that the substitution $G[\lambda x, y \,.\, q(y)/n]$ yields the process-equation

$$q(y) \quad = \quad a \cdot q(g(y)) + a.$$

We derive:

$$\dfrac{\dfrac{\overline{G[\lambda x, y \,.\, p(x)/n] \ \mathbf{from}\ E, \{\langle x : S\rangle\}, \emptyset}}{G[\lambda x, y \,.\, p(x)/n] \ \mathbf{from}\ E, \{\langle x : S\rangle, \langle y : S\rangle\}, \emptyset} \qquad \dfrac{\dfrac{\overline{q(x) = a \cdot q(g(x)) + a \ \mathbf{from}\ E, \{\langle x : S\rangle\}, \emptyset}}{q(x) = a \cdot q(g(x)) + a, \ \mathbf{from}\ E, \{\langle x : S\rangle, \langle y : S\rangle\}, \emptyset}}{G[\lambda x, y \,.\, q(y)/n] \ \mathbf{from}\ E, \{\langle x : S\rangle, \langle y : S\rangle\}, \emptyset}}{p(x) = q(y) \ \mathbf{from}\ E, \{\langle x : S\rangle, \langle y : S\rangle\}, \emptyset} \ \text{RSP}$$

(End example.)

4.4 Soundness

In this section we argue that the proof system presented here is *sound*, i.e. that all properties derivable by the axioms and rules introduced thus far are valid in any appropriate semantical setting (see definition 3.3.1). We can express this as

$$\mathcal{M} \vdash \phi \ \mathbf{from}\ E, V_d, V_p \implies \mathbb{A}, \approx_{\mathbb{A}} \models \phi \ \mathbf{from}\ E, V_d, V_p$$

for $\mathcal{M} = \text{BOOL}+\text{FACT}+\text{IND}(\overline{C})+\text{REC}+\text{ACP}+\text{SC}+\text{HIDE}+\text{REN}+\text{COND}+\text{SUM}+\text{RSP}$.

We first present the modules $\text{IND}(\overline{C})$ and RSP in an *axiomatic* style to establish their validity apart from the soundness of the rules for natural deduction. This is more comprehensible, and it is closer to the literature. We rephrase $\text{IND}(\overline{C})$ as

$$\overline{\text{IND}}(\overline{C}) \equiv \dfrac{}{\bigwedge_{i=1}^{m}\Big(\bigwedge_{j=1}^{k_i}\Big(\bigwedge_{\sigma \in I_{ij}} \sigma(\phi) \to \phi[f_{ij}(z_1^{ij}, \ldots, z_{l_{ij}}^{ij})/x_i]\Big)\Big) \to \phi \ \mathbf{from}\ E, V_d \cup V_z, V_p}$$

where V_z is the union of all the sets $\{\langle z_n^{ij} : S_n^{ij}\rangle \mid 1 \leq n \leq l_{ij}\}$, and we rephrase RSP in a similar way:

$$\overline{\text{RSP}} \equiv \dfrac{}{\bigwedge_{i=1}^{m} G_i[\lambda \bar{x}_j \,.\, p_j(\bar{x}_j)/n_j]_{j=1}^{m} \wedge G_i[\lambda \bar{x}_j \,.\, q_j(\bar{x}_j)/n_j]_{j=1}^{m} \to p_k(\bar{x}_k) = q_k(\bar{x}_k) \ \mathbf{from}\ E, V_d, V_p}$$

for $1 \leq k \leq m$. Note that these formulations are indeed logically equivalent. For the module SUM we define SUM^- by omitting the rule SUM11 (the congruence rule for the sum operator).

Let in the rest of this section E be a specification, $\mathbb{A}$ be a model of E and $\approx_{\mathbb{A}} \supseteq \ \leftrightarrows_{\mathbb{A}}$ be a congruence of process terms that is representation insensitive. Let furthermore $\overline{\mathcal{M}}$ contain all the modules presented thus far, where IND and RSP are replaced by their axiomatic counterparts, and SUM is replaced by SUM^-. We now argue that all axioms in $\overline{\mathcal{M}}$ are valid.

As for the modules for data equivalence we have that the validity of BOOL follows from the fact that $\mathbb{A}$ is boolean preserving and the validity of FACT follows immediately by definition 3.3.1. We give a short argument for the validity of the axiom $\overline{\mathrm{IND}}(\overline{C})$: Let

$$P \equiv \bigwedge_{i=1}^{m}(\bigwedge_{j=1}^{k_i}(\bigwedge_{\sigma\in I_{ij}}\sigma(\phi)\rightarrow\phi[f_{ij}(z_1^{ij},...,z_{l_{ij}}^{ij})/x_i]))$$

and assume that $\mathbb{A},\approx_{\mathbb{A}}\ \models P$ **from** $E, V_d\cup V_z, V_p$ and (for simplicity) that ϕ contains no process variables. Further assume that any data variable occurring in ϕ is among data variables $x_1,...,x_l$ (the induction takes place on the variables $x_1,...,x_m$ for some $m\leq l$). It is sufficient to show that $\mathbb{A},\approx_{\mathbb{A}}\ \models \phi[t_1,...,t_l/x_1,...,x_l]$ for arbitrary closed data terms t_i (the notation here expresses the simultaneous substitution of t_i for x_i). By $\overline{C}$ consisting of constructor sets and $\mathbb{A}$ being a minimal algebra we may assume that $t_1,...,t_m$ only contain constructor elements. We apply structural induction on the total complexity of the terms $t_1,...,t_m$.

1. None of $t_1,...,t_m$ consist of a constructor function *applied* to terms of one of the sorts of $x_1,...x_m$. In this case each of the index sets I_{ij} is empty, so all these conjunctions are satisfied, and hence

$$\mathbb{A},\approx_{\mathbb{A}}\ \models\bigwedge_{i=1}^{m}(\bigwedge_{j=1}^{k_i}\phi[f_{ij}(z_1^{ij},...,z_{l_{ij}}^{ij})/x_i])$$

   by assumption. As $t_1,...,t_m$ are applications of one of the constructor functions, it follows that $\phi[t_1,...,t_l/x_1,...,x_l]$ is valid in $\mathbb{A},\approx_{\mathbb{A}}$.

2. It is not the case that 1 holds. Consider some t_i of the form $f_{ij}(s_1,...,s_{l_{ij}})$ such that I_{ij} is not empty. By assumption we have that

$$\mathbb{A},\approx_{\mathbb{A}}\ \models\rho(\bigwedge_{\sigma\in I_{ij}}\sigma(\phi)\rightarrow\phi[f_{ij}(z_1^{ij},...,z_{l_{ij}}^{ij})/x_i]) \tag{3}$$

   where ρ is the data substitution that maps z_n^{ij} to s_n for $n=1\ldots l_{ij}$ and x_k to t_k for $k=1\ldots l$. Since all the conjuncts $\rho(\sigma(\phi))$ (there is at least one such a conjunct) yield a strictly lower total complexity than $t_1,...,t_m$, we have by the induction hypothesis that all these are valid. By (3) it then follows that $\phi[t_1,...,t_l/x_1,...,x_l]$ is valid in $\mathbb{A},\approx_{\mathbb{A}}$.

With respect to the validity of the axioms in the modules ACP, SC, HIDE and REN for process equivalence we refer to the standard literature [BW90, Gla90]. The validity of the modules SUM$^-$ and COND follows trivially. For an idea of a soundness proof for RSP see also [BW90]. That the guardedness of the system G in RSP is a *necessary* condition can be easily seen from the case in which G is a system containing equations of the form $n=n$ or $n(x_1,...,x_m)=n(x_1,...,x_m)$: in this case we can prove any two processes terms identical with our formulation of RSP.

We conclude with a general argument for the soundness of our proof system. We write $\mathbb{A},\approx_{\mathbb{A}}\ \models\Gamma$ for Γ a set of property formulas if $\mathbb{A},\approx_{\mathbb{A}}\ \models\theta$ for each $\theta\in\Gamma$, so $\mathbb{A},\approx_{\mathbb{A}}\ \models\emptyset$ by default. As to deal with *cancellation* of open assumptions, we split the argument into three steps.

Step 1. We first prove a result concerning applications of the rule VAR:

$$\phi \textbf{ from } E, V'_d, V'_p \vdash \phi \textbf{ from } E, V_d, V_p \text{ by using only VAR} \implies$$

$$(\mathbb{A}, \approx_{\mathbb{A}} \models \phi \textbf{ from } E, V'_d, V'_p \Longrightarrow \mathbb{A}, \approx_{\mathbb{A}} \models \phi \textbf{ from } E, V_d, V_p).$$

This follows easy by structural induction on ϕ. In particular this implies that any substitution instance of one of the axioms of $\overline{\mathcal{M}}$ (over proper variable sets) is valid in $\mathbb{A}, \approx_{\mathbb{A}}$.

Step 2. Any deduction with conclusion ϕ **from** E, V_d, V_p can be converted into a corresponding deduction over uniform $V'_d \supseteq V_d$ and $V'_p \supseteq V_p$ with conclusion ϕ **from** E, V'_d, V'_p. This can be done by using related open assumptions and 'derived' axioms over V'_d, V'_p (see step 1). We use the notation $\overline{\mathcal{M}}(V'_d, V'_p)$ for the latter. We call such deductions *uniform*. So uniform deductions do not contain applications of the rule VAR.

Step 3. Let V_d, V_p be fixed. We now only consider property formulas over E, V_d, V_p and further omit this attribute. We write

$$\overline{\mathcal{M}}(V_d, V_p), \Gamma \vdash_{\ uniform} \phi$$

iff there is a uniform deduction with all uncancelled hypotheses in Γ.

Now the proof system $\vdash_{uniform}$ can be proved sound in the standard way (cf. [Dal83]). To be precise: let DS be the set of ground data substitutions over E, V_d, V_p and PS be the set of ground process substitutions over E, V_d, V_p. Then

$$\overline{\mathcal{M}}(V_d, V_p) + \Gamma \vdash_{uniform} \phi \implies \forall \sigma' \in DS \;\; \forall \sigma \in PS$$

$$(\mathbb{A}, \approx_{\mathbb{A}} \models \sigma'(\sigma(\Gamma)) \Longrightarrow \mathbb{A}, \approx_{\mathbb{A}} \models \sigma'(\sigma(\phi)) \textbf{ from } E, V_d, V_p).$$

This can be proved by induction on the length of derivations. As an example we show this for application of the rule SUM11.

Example 4.4.1. Assume

$$\frac{\begin{array}{c}\Gamma\\ \mathcal{D}\\ p = q\end{array}}{\Sigma(d : D, p) = \Sigma(d : D, q)} \; \text{SUM11}$$

(so d occurs not free in any hypothesis in Γ). Now suppose

$$\mathbb{A}, \approx_{\mathbb{A}} \models \sigma'(\sigma(\Gamma)) \tag{4}$$

for some $\sigma \in PS$ and $\sigma' \in DS$. We must show that

$$\mathbb{A}, \approx_{\mathbb{A}} \models \sigma'(\sigma(\Sigma(d : D, p))) = \sigma'(\sigma(\Sigma(d : D, q))). \tag{5}$$

This is the case if

$$\mathbb{A}, \approx_{\mathbb{A}} \models \Sigma(e : D, \sigma''(\sigma(p[e/d]))) = \Sigma(e : D, \sigma''(\sigma(q[e/d]))) \tag{6}$$

with σ'' as σ' and the fresh variable e is mapped to itself. Note that e is the only variable that can occur free in $\sigma''(\sigma(p[e/d]))$ and $\sigma''(\sigma(q[e/d]))$. Now 6 holds by definition iff for any ground data term t of sort D it holds that

$$\mathbb{A}, \approx_{\mathbb{A}} \models \sigma''(\sigma(p[e/d]))[t/e] = \sigma''(\sigma(q[e/d]))[t/e]$$

which is by definition of σ'' and σ the case iff

$$\mathbb{A}, \approx_{\mathbb{A}} \models \sigma'(\sigma(p)[t/d]) = \sigma'(\sigma(q)[t/d]). \tag{7}$$

Fix such a substitution $[t/d]$. As d is not free in Γ, it follows from 4 that

$$\mathbb{A}, \approx_{\mathbb{A}} \models \sigma'(\sigma(\Gamma)[t/d]).$$

Because $\sigma' \circ [t/d] \in DS$, it follows by induction that 7 holds, and hence also 5.
(End example.)

So taking $\Gamma = \emptyset$ it follows that

$$\overline{\mathcal{M}}(V_d, V_p) \vdash_{uniform} \phi \implies \mathbb{A}, \approx_{\mathbb{A}} \models \phi.$$

Combination of this result with steps 1, 2 and the logical equivalence of $\overline{\mathcal{M}}$ and $\mathcal{M} \setminus \{\text{SUM11}\}$ finally yields

$$\mathcal{M} \vdash \phi \textbf{ from } E, V_d, V_p \implies \mathbb{A}, \approx_{\mathbb{A}} \models \phi \textbf{ from } E, V_d, V_p.$$

5 Examples

In this section we provide some examples of proofs in μCRL. As formal proofs (i.e. deductions) of non-trivial facts are often hard to read, and may take in our case a larger space than available on one page, we will not give these. Instead we only write down the essential steps of a proof, trusting that the suggestion of a formal proof is sufficiently clear.

Furthermore we will often use the symbol $=$ (possibly superscripted with some names) to represent proofs in a linear style: in a context where E, V_d, V_p are fixed, we write

$$t = u$$

if this identity can be obtained by applications of reflexivity, symmetry or substitutivity (see lemma 4.1.6.1+2+4), or via the rule SUB (so no variables that occur free in an open assumption are instantiated). Moreover, based on the transitivity of $=$, proved in lemma 4.1.6.3, we write

$$t_1 = t_2 = \ldots = t_n$$

to represent a proof with conclusion $t_1 = t_n$. Sometimes, when it is clear how to prove $t_1 = t_2 = \ldots = t_n$, we only write down $t_1 = t_n$. For convenience we generally write names of axioms or identities above the $=$.

5.1 Another application of RSP

Consider the following guarded specification:

$$
\begin{array}{lll}
E \equiv & \textbf{sort} & \textbf{Bool} \\
& \textbf{func} & T, F :\rightarrow \textbf{Bool} \\
& & \neg : \textbf{Bool} \rightarrow \textbf{Bool} \\
& \textbf{rew} & \neg(T) = F \\
& & \neg(F) = T \\
& & \\
& \textbf{sort} & Nat \\
& \textbf{func} & 0 :\rightarrow Nat \\
& & s : Nat \rightarrow Nat \\
& & even : Nat \rightarrow \textbf{Bool} \\
& \textbf{var} & x : Nat \\
& \textbf{rew} & even(0) = T \\
& & even(s(x)) = \neg(even(x)) \\
& & \\
& \textbf{act} & a : Nat \\
& \textbf{proc} & p(x : Nat) = a(even(x)) \cdot p(s(x)) \\
& & q(b : \textbf{Bool}) = a(b) \cdot q(\neg(b))
\end{array}
$$

With RSP we can show that $p(x) = q(even(x))$ **from** $E, \{\langle x : Nat\rangle\}, \emptyset$. To that end we define

$$G \stackrel{def}{=} n(x) = a(even(x)) \cdot n(s(x))$$

so that $E(G)$ is guarded. Because $G[\lambda x \,.\, p(x)/n] \equiv p(x) = a(even(x)) \cdot p(s(x))$ it is obvious that

$$G[\lambda x \,.\, p(x)/n] \ \textbf{from} \ E, \{\langle x : \text{Nat}\rangle\}, \emptyset$$

is derivable by the axiom REC. In order to apply RSP we have to derive

$$G[\lambda x \,.\, q(even(x))/n] \equiv q(even(x)) = a(even(x)) \cdot q(even(s(x))).$$

This can be done as follows:

$$
\dfrac{\dfrac{\dfrac{}{q(b) = a(b) \cdot q(\neg(b))}\ \text{REC}}{\begin{array}{c}\vdots \\ (\text{VAR, SUB}) \\ \vdots \\ q(even(x)) = a(even(x)) \cdot q(\neg(even(x)))\end{array}} \qquad \dfrac{\dfrac{}{even(s(x)) = \neg(even(x))}\ \text{FACT}}{\neg(even(x)) = even(s(x))}\ 4.1.3}{q(even(x)) = a(even(x)) \cdot q(even(s(x)))}\ \text{REPL}
$$

Now RSP gives that $p(x) = q(even(x))$ **from** $E, \{\langle x : Nat\rangle\}, \emptyset$.

5.2 Proving some properties of bags

We give a specification of a process that behaves like a bag and prove some properties about it. The process Bag can input data and it can output data that have been put in the bag before. The bag itself is described as a data type Bag. It has the usual operations such as $\emptyset$, in, rem, $test$ and con for the empty bag and the input, remove, test and concatenate operators. The process uses the actions r and s to read and send data from and to the environment. The specification BAG is defined in table 8, where we added names to its axioms for easy reference. In this specification it is left open how the data are specified. We only assume the presence of an equality function eq, which is partly specified. The other functions in BAG are specified in a straightforward way. Note that it is not hard to check that

$$C \stackrel{def}{=} \{\emptyset :\to Bag, in : D \times Bag \to Bag\}$$

is a set of constructors of sort Bag, and that BAG is a guarded specification.

Lemma 5.2.1. *Let $V_d = \{\langle d : D\rangle, \langle b : Bag\rangle, \langle c : Bag\rangle\}$. We have the following useful facts about bags:*

1. $\text{IND}(C) \vdash con(in(d,b),c) = con(b, in(d,c))$ **from** $BAG, V_d, \emptyset,$
2. $\text{IND}(C) \vdash con(b,c) = con(c,b)$ **from** $BAG, V_d, \emptyset,$
3. $\text{BOOL} + \text{IND}(C) \vdash$
   $rem(d, con(b,c)) \lhd test(d,b) \rhd \delta = con(rem(d,b),c) \lhd test(d,b) \rhd \delta$ **from** $BAG, V_d, \emptyset,$
4. $\text{ACP} + \text{BOOL} + \text{COND} + \text{IND}(C) \vdash$
   $x \lhd test(d, con(b,c)) \rhd \delta = x \lhd test(d,b) \rhd \delta + x \lhd test(d,c) \rhd \delta$ **from** $BAG, V_d, \{x\}.$

Proof.
Ad 1. We prove this by induction on the variable b, so we must prove 1 for both $b = \emptyset$ and $b = in(e,b')$ over $V_d \cup \{\langle e : D\rangle, \langle b' : Bag\rangle\}$.
Suppose $b = \emptyset$. Then

$$con(in(d,\emptyset),c) \stackrel{\text{BAG7}}{=} in(d, con(\emptyset,c)) \stackrel{\text{BAG6}}{=} in(d,c) \stackrel{\text{BAG6}}{=} con(\emptyset, in(d,c)).$$

Suppose $b = in(e,b')$ and assume that

$$con(in(d,b'),c) = con(b', in(d,c)). \tag{8}$$

Then

$$\begin{aligned} con(in(d,in(e,b')),c) &\stackrel{\text{BAG7}}{=} in(d, con(in(e,b'),c)) \stackrel{\text{BAG7}}{=} in(d, in(e, con(b',c))) \\ &\stackrel{\text{BAG3}}{=} in(e, in(d, con(b',c))) \stackrel{\text{BAG7}}{=} in(e, con(in(d,b'),c)) \\ &\stackrel{(8)}{=} in(e, con(b', in(d,c))) \stackrel{\text{BAG7}}{=} con(in(e,b'), in(d,c)). \end{aligned}$$

By $\text{IND}(C)$ we conclude that 1 holds.

sort	**Bool**	
func	$T, F :\to \mathbf{Bool}$	
	$\mathit{if} : \mathbf{Bool} \times \mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool}$	
var	$b_1, b_2 : \mathbf{Bool}$	
rew	$\mathit{if}(T, b_1, b_2) = b_1$	IF1
	$\mathit{if}(F, b_1, b_2) = b_2$	IF2
sort	D	
func	...	
	$eq : D \times D \to \mathbf{Bool}$	
var	...	
	$d : D$	
rew	...	
	$eq(d, d) = T$	EQ
sort	Bag	
func	$\emptyset :\to Bag$	
	$\mathit{if} : \mathbf{Bool} \times Bag \times Bag \to Bag$	
	$test : D \times Bag \to \mathbf{Bool}$	
	$in, rem : D \times Bag \to Bag$	
	$con : Bag \times Bag \to Bag$	
var	$d, e : D$	
	$b, c : Bag$	
rew	$\mathit{if}(T, b, c) = b$	IF3
	$\mathit{if}(F, b, c) = c$	IF4
	$test(d, \emptyset) = F$	BAG1
	$test(d, in(e, b)) = \mathit{if}(eq(d, e), T, test(d, b))$	BAG2
	$in(d, in(e, b)) = in(e, in(d, b))$	BAG3
	$rem(d, \emptyset) = \emptyset$	BAG4
	$rem(d, in(e, b)) = \mathit{if}(eq(d, e), b, in(e, rem(d, b)))$	BAG5
	$con(\emptyset, b) = b$	BAG6
	$con(in(d, b), c) = in(d, con(b, c))$	BAG7
act	$r, s : D$	
proc	$Bag(x : Bag) = \Sigma(d : D, r(d) \cdot Bag(in(d, x))) +$ $\Sigma(d : D, s(d) \cdot Bag(rem(d, x)) \triangleleft test(d, x) \triangleright \delta)$	

Table 8: The specification BAG

Ad 2. We prove statement 2 of the lemma again by induction on b.
Suppose $b = \emptyset$. We first prove $con(\emptyset, c) = con(c, \emptyset)$ by induction on c.

Suppose $c = \emptyset$. Then $con(\emptyset, \emptyset) = con(\emptyset, \emptyset)$.
Suppose $c = in(d, c')$ and assume

$$con(\emptyset, c') = con(c', \emptyset). \tag{9}$$

Then $con(\emptyset, in(d, c')) \overset{\text{BAG6}}{=} in(d, c') \overset{(9)}{=} in(d, con(c', \emptyset)) \overset{\text{BAG7}}{=} con(in(d, c'), \emptyset)$. By IND$(C)$ it follows that $con(\emptyset, c) = con(c, \emptyset)$.

Suppose $b = in(d, b')$ and assume

$$con(b', c) = con(c, b'). \tag{10}$$

Then

$$\begin{aligned} con(in(d, b'), c) &\overset{\text{BAG7}}{=} in(d, con(b', c)) \overset{(10)}{=} in(d, con(c, b')) \overset{\text{BAG7}}{=} con(in(d, c), b') \\ &\overset{5.2.1.1}{=} con(c, in(d, b')). \end{aligned}$$

By IND(C) it follows that 2 holds.

Ad 3. We prove this by induction on the variable b.
Suppose $b = \emptyset$. Then $test(d, \emptyset) \overset{\text{BAG1}}{=} F$. Hence we can derive 3.
Suppose $b = in(e, b')$. Assume that

$$rem(d, con(b', c)) \triangleleft test(d, b') \triangleright \delta = con(rem(d, b'), c) \triangleleft test(d, b') \triangleright \delta. \tag{11}$$

Further assume that $test(d, in(e, b')) = T$ (otherwise we are done). We have that

$$\begin{aligned} rem(d, con(in(e, b'), c)) &\overset{\text{BAG7}}{=} rem(d, in(e, con(b', c))) \\ &\overset{\text{BAG5}}{=} if(eq(d, e), con(b', c), in(e, rem(d, con(b', c))))). \end{aligned} \tag{12}$$

Suppose $eq(d, e) = T$. Then $(12) \overset{\text{IF3}}{=} con(b', c) \overset{\text{IF3,BAG5}}{=} con(rem(d, in(e, b')), c)$. So we conclude via $\rightarrow$I that in this case 3 holds.

Now suppose $eq(d, e) = F$. So

$$(12) \overset{\text{IF4}}{=} in(e, rem(d, con(b', c))) \tag{13}$$

As $eq(d, e) = F$ it holds that $test(d, in(e, b')) \overset{\text{BAG2,IF2}}{=} test(d, b')$ which is equal to T by assumption. By (11) we conclude that $(13) = in(e, con(rem(d, b'), c))$. Hence also in this case 3 holds. By IND(C) it follows that 3 holds in general.

Ad 4. This proof is again carried out by induction on b.
Suppose $b = \emptyset$. Then

$$\begin{aligned} x \lhd test(d, con(\emptyset, c)) \rhd \delta &\stackrel{\text{BAG6,A6,A1}}{=} \delta + x \lhd test(d, c) \rhd \delta \\ &\stackrel{\text{Cond2}}{=} x \lhd F \rhd \delta + x \lhd test(d, c) \rhd \delta \\ &\stackrel{\text{BAG1}}{=} x \lhd test(d, \emptyset) \rhd \delta + x \lhd test(d, c) \rhd \delta. \end{aligned}$$

Now suppose $b = in(e, b')$. Assume that

$$x \lhd test(d, con(b', c)) \rhd \delta \quad = \quad x \lhd test(d, b') \rhd \delta + x \lhd test(d, c) \rhd \delta \tag{14}$$

This implies that

$$\begin{aligned} x \lhd test(d, con(in(e, b'), c)) \rhd \delta &\stackrel{\text{BAG7}}{=} x \lhd test(d, in(e, con(b', c))) \rhd \delta \\ &\stackrel{\text{BAG2}}{=} x \lhd if(eq(d, e), T, test(d, con(b', c))) \rhd \delta \end{aligned} \tag{15}$$

Assume $eq(d, e) = T$. Then

$$\begin{aligned} (15) &\stackrel{\text{IF1}}{=} x \lhd T \rhd \delta \stackrel{\text{Cond1}}{=} x \\ &\stackrel{\text{lemma 4.3.1.2,Cond1}}{=} x \lhd T \rhd \delta + x \lhd test(d, c) \rhd \delta \\ &\stackrel{\text{IF1,BAG2}}{=} x \lhd test(d, in(e, b')) \rhd \delta + x \lhd test(d, c) \rhd \delta. \end{aligned}$$

Now assume $eq(d, e) = F$.

$$\begin{aligned} (15) &\stackrel{\text{IF2}}{=} x \lhd test(d, con(b', c)) \rhd \delta \\ &\stackrel{(14)}{=} x \lhd test(d, b') \rhd \delta + x \lhd test(d, c) \rhd \delta \\ &\stackrel{\text{IF2,BAG2}}{=} x \lhd test(d, in(e, b')) \rhd \delta + x \lhd test(d, c) \rhd \delta. \end{aligned}$$

Using B2 we conclude that 4 holds for $b = in(e, b')$. By IND(C) it follows that 4 holds in general. □

There is the following relation between the parallel operator and the concatenation operator *con* on the data type *Bag*.

Theorem 5.2.2. *Two bags in parallel behave as one bag with the contents concatenated. Let* $V_d = \{\langle b : Bag\rangle, \langle c : Bag\rangle\}$. *Then*

$$\text{ACP} + \text{BOOL} + \text{COND} + \text{IND}(C) + \text{RSP} + \text{SUM} \vdash$$

$$Bag(b) \parallel Bag(c) = Bag(con(b, c)) \ \ \textbf{from} \ BAG, V_d, \emptyset.$$

Proof. The main step in the proof is an application of RSP. First we define the system G as follows:

$$\begin{aligned} G \stackrel{def}{=} n(b, c) \quad = \quad & \Sigma(d : D, r(d) \cdot n(in(d, b), c)) + \\ & \Sigma(d : D, r(d) \cdot n(b, in(d, c))) + \\ & \Sigma(d : D, s(d) \cdot n(rem(d, b), c) \lhd test(d, b) \rhd \delta) + \\ & \Sigma(d : D, s(d) \cdot n(b, rem(d, c)) \lhd test(d, c) \rhd \delta). \end{aligned}$$

Observe that G is guarded.

We prove $G[\lambda b, c\ .\ Bag(b) \parallel Bag(c)/n]$ **from** $BAG, V_d \cup \{\langle d : D\rangle\}, \emptyset$ and $G[\lambda b, c\ .\ Bag(con(b, c))/n]$ **from** $BAG, V_d \cup \{\langle d : D\rangle\}, \emptyset$. Then by RSP and VAR the theorem follows in a straightforward way.

First we show $G[\lambda b, c\ .\ Bag(b) \parallel Bag(c)/n]$. This a straightforward expansion.

$$\begin{aligned}
Bag(b) \parallel Bag(c) \overset{\text{expansion}}{=}\ & \Sigma(d : D, r(d) \cdot (Bag(in(d, b)) \parallel Bag(c)))+\\
& \Sigma(d : D, r(d) \cdot (Bag(b) \parallel Bag(in(d, c))))+\\
& \Sigma(d : D, s(d) \cdot (Bag(rem(d, b)) \parallel Bag(c)) \lhd test(d, b) \rhd \delta)+\\
& \Sigma(d : D, s(d) \cdot (Bag(b) \parallel Bag(rem(d, c))) \lhd test(d, c) \rhd \delta).
\end{aligned}$$

Now we show $G[\lambda b, c\ .\ Bag(con(b, c))/n]$. Lemma 5.2.1 turns out to be handy.

$$\begin{aligned}
Bag(con(b, c)) \overset{\text{expansion}}{=}\ & \Sigma(d : D, r(d) \cdot Bag(in(d, con(b, c))))+\\
& \Sigma(d : D, s(d) \cdot Bag(rem(d, con(b, c))) \lhd test(d, con(b, c)) \rhd \delta)\\
\overset{(1)}{=}\ & \Sigma(d : D, r(d) \cdot Bag(con(in(d, b), c)))+\\
& \Sigma(d : D, r(d) \cdot Bag(con(b, in(d, c))))+\\
& \Sigma(d : D, s(d) \cdot Bag(rem(d, con(b, c))) \lhd test(d, b) \rhd \delta)+\\
& \Sigma(d : D, s(d) \cdot Bag(rem(d, con(c, b))) \lhd test(d, c) \rhd \delta)\\
\overset{(2)}{=}\ & \Sigma(d : D, r(d) \cdot Bag(con(in(d, b), c)))+\\
& \Sigma(d : D, r(d) \cdot Bag(con(b, in(d, c))))+\\
& \Sigma(d : D, s(d) \cdot Bag(con(rem(d, b), c)) \lhd test(d, b) \rhd \delta)+\\
& \Sigma(d : D, s(d) \cdot Bag(con(b, rem(d, c))) \lhd test(d, c) \rhd \delta)
\end{aligned}$$

where (1) follows from A3, lemma 5.2.1.1+2+4 and SUM11, and (2) from lemma 5.2.1.3, lemma 4.3.1.1 and SUM11. □

Corollary 5.2.3. *Let* $V_d = \{\langle a : Bag\rangle, \langle b : Bag\rangle, \langle c : Bag\rangle\}$. *Then*

$$\text{ACP} + \text{BOOL} + \text{COND} + \text{IND}(C) + \text{RSP} + \text{SUM} \vdash$$

$$(Bag(a) \parallel Bag(b)) \parallel Bag(c) = Bag(a) \parallel (Bag(b) \parallel Bag(c)) \textbf{ from } BAG, V_d, \emptyset.$$

Proof. By associativity of the concatenation operator *con* (easy) and theorem 5.2.2. □

This corollary is of interest as it is a non-closed instance of

$$(x \parallel y) \parallel z = x \parallel (y \parallel z),$$

an identity that is not derivable from ACP (but follows from ACP + SC).

We conclude with the following theorem, stating that the process specified by $Bag(\emptyset)$ satisfies a standard definition (see [BK84a, BW90], though there the sort D has to be finite).

Theorem 5.2.4. *The process* $Bag(\emptyset)$ **from** BAG *satisfies*

$$\mathcal{M} \vdash Bag(\emptyset) = \Sigma(d : D, r(d) \cdot (Bag(\emptyset) \parallel s(d))) \textbf{ from } BAG, \emptyset, \emptyset$$

provided $\mathcal{M} \supseteq \{\text{ACP}, \text{BOOL}, \text{COND}, \text{IND}(C), \text{RSP}, \text{SUM}\}$ *is such that we can derive* $\mathcal{M} \vdash eq(d, e) = T \rightarrow d = e$ **from** $BAG, \{\langle d : D\rangle, \langle e : D\rangle\}, \emptyset$.

Proof. Let $V_d = \{\langle d : D\rangle, \langle e : D\rangle, \langle b : Bag\rangle\}$. We first establish an intermediate result in three steps:

1. $test(e, b) = T \rightarrow rem(e, in(d, b)) = in(d, rem(e, b))$ **from** $BAG, V_d, \emptyset$.

   This can be proved by induction on the variable b (cf. the proof of lemma 5.2.1.3).

2. $s(d) \cdot Bag(b) + \delta \lhd eq(e, d) \rhd (s(e) \cdot Bag(rem(e, in(d, b))) \lhd test(e, in(d, b)) \rhd \delta) =$
   $s(d) \cdot Bag(b) + s(e) \cdot Bag(in(d, rem(e, b))) \lhd test(e, b) \rhd \delta$ **from** $BAG, V_d, \emptyset$.

   This can be proved by case distinction of the four possible values of $eq(e, d)$ and $test(e, b)$. In case both these are true, we need the property of the equality function eq, as we must derive that $s(d) \cdot Bag(b) = s(d) \cdot Bag(b) + s(e) \cdot Bag(in(d, rem(e, b)))$. In the case that $eq(e, d) = F$ and $test(e, b) = T$ we need 1 above.

3. $s(d) \cdot Bag(b) + \Sigma(e : D, \delta \lhd eq(e, d) \rhd (s(e) \cdot Bag(rem(e, in(d, b))) \lhd test(e, in(d, b)) \rhd \delta)) =$
   $s(d) \cdot Bag(b) + \Sigma(e : D, s(e) \cdot Bag(in(d, rem(e, b))) \lhd test(e, b) \rhd \delta)$ **from** $BAG, V_d, \emptyset$.

   This follows from 2 by SUM.

With result 3 and RSP we can show that $Bag(b) \parallel s(d) = Bag(in(d, b))$ **from** $BAG, V_d, \emptyset$. This identity plays a crucial role in the proof of the theorem. Let the system G be defined as follows:

$$\begin{aligned} G \stackrel{def}{=} n(d, b) \quad = \quad & \Sigma(e : D, r(e) \cdot n(d, in(e, b)))) \; + \\ & \Sigma(e : D, s(e) \cdot n(d, rem(e, b)) \lhd test(e, b) \rhd \delta) \; + \\ & s(d) \cdot Bag(b) \end{aligned}$$

so that G is a guarded system. We can derive $G[\lambda d, b \,.\, Bag(b) \parallel s(d)/n]$ by a straightforward expansion. We give a derivation of $G[\lambda d, b \,.\, Bag(in(d, b))/n]$:

$$\begin{aligned} Bag(in(d, b)) \quad & \stackrel{\text{expansion}}{=} && \Sigma(e : D, r(e) \cdot Bag(in(e, in(d, b)))) \; + \\ & && \Sigma(e : D, s(e) \cdot Bag(rem(e, in(d, b))) \lhd test(e, in(d, b)) \rhd \delta) \\ & \stackrel{4.3.2}{=} && \Sigma(e : D, r(e) \cdot Bag(in(e, in(d, b)))) \; + \\ & && \Sigma(e : D, \delta \lhd eq(e, d) \rhd \\ & && \quad (s(e) \cdot Bag(rem(e, in(d, b))) \lhd test(e, in(d, b)) \rhd \delta)) \; + \\ & && s(d) \cdot Bag(rem(d, in(d, b))) \lhd test(d, in(d, b)) \rhd \delta \\ & \stackrel{\text{EQ,BAG2+3+5}}{=} && \Sigma(e : D, r(e) \cdot Bag(in(d, in(e, b)))) \; + \\ & && \Sigma(e : D, \delta \lhd eq(e, d) \rhd \\ & && \quad (s(e) \cdot Bag(rem(e, in(d, b))) \lhd test(e, in(d, b)) \rhd \delta)) \; + \\ & && s(d) \cdot Bag(b) \\ & \stackrel{\text{result3}}{=} && \Sigma(e : D, r(e) \cdot Bag(in(d, in(e, b)))) \; + \\ & && \Sigma(e : D, s(e) \cdot Bag(in(d, rem(e, b))) \lhd test(e, b) \rhd \delta) \; + \\ & && s(d) \cdot Bag(b) \end{aligned}$$

By RSP it follows that $Bag(b) \parallel s(d) = Bag(in(d, b))$, and hence

$$r(d) \cdot (Bag(\emptyset) \parallel s(d)) = r(d) \cdot Bag(in(d, \emptyset)) \textbf{ from } BAG, V_d, \emptyset.$$

By SUM11 and VAR it then follows that

$$\Sigma(d : D, r(d) \cdot (Bag(\emptyset) \parallel s(d))) = \Sigma(d : D, r(d) \cdot Bag(in(d,\emptyset))) \textbf{ from } BAG, \emptyset, \emptyset$$

and as $\Sigma(d : D, r(d) \cdot Bag(in(d,\emptyset))) = Bag(\emptyset)$ **from** $BAG, \emptyset, \emptyset$ this concludes the proof. $\square$

References

[Apt81] K.R. Apt. Ten years of Hoare's logic, a survey, part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[Apt84] K.R. Apt. Ten years of Hoare's logic, a survey, part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.

[BB90] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. Report P9008, University of Amsterdam, Amsterdam, 1990. To appear in *Proceedings NATO ASI Summer School*, Marktoberdorf 1990, LNCS.

[BK84a] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proceedings* 11^{th} *ICALP,* Antwerp, volume 172 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 1984.

[BK84b] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

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

[CCI87] CCITT Working Party X/1. *Recommendation Z.100 (SDL)*, 1987.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[Dal83] D. van Dalen. *Logic and Structure.* Springer-Verlag, 1983.

[DS90] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*, volume 14 of *Texts and Monographs in Computer Science.* Springer-Verlag, 1990.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications I*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[Gla90] R.J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions.* PhD thesis, Free University, Amsterdam, 1990.

[GP90] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Report CS-R9076, CWI, Amsterdam, 1990.

[GP91] J.F. Groote and A. Ponse. μCRL: A base for analysing processes with data. In E. Best and G. Rozenberg, editors, *Proceedings* $3^{\rm rd}$ *Workshop on Concurrency and Compositionality, Goslar, GMD-Studien Nr. 191*, pages 125–130. Universität Hildesheim, 1991.

[GTL89] J-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[GV89] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence (extended abstract). In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings* 16^{th} *ICALP,* Stresa, volume 372 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 1989. Full version to appear in *Information and Computation.*

[HHJ+87] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[HI90] M. Hennessy and A. Ingólfsdóttir. A theory of communicating processes with value-passing. In M.S. Paterson, editor, *Proceedings* 17^{th} *ICALP,* Warwick, volume 443 of *Lecture Notes in Computer Science*, pages 209–219. Springer-Verlag, 1990.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[ISO87] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I + II. Technical Report ECS-LFCS-89-85 + ECS-LFCS-89-86, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989. To appear in *Information and Computation.*

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[Par81] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[Ss90] SPECS-semantics. *Definition of MR and CRL Version 2.1*, 1990.

[Sza69] M.E. Szabo. *The Collected Papers of Gerhard Gentzen.* North-Holland, 1969.

[TD88] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction (vol I).* North-Holland, 1988.