Process Algebra in PVS

Twan Basten[1] and Jozef Hooman[2]

[1] Dept. of Computing Science, Eindhoven University of Technology, The Netherlands
tbasten@win.tue.nl, http://www.win.tue.nl/∼tbasten
[2] Computing Science Institute, University of Nijmegen, The Netherlands
hooman@cs.kun.nl, http://www.cs.kun.nl/∼hooman

Abstract. The aim of this work is to investigate mechanical support for process algebra, both for concrete applications and theoretical properties. Two approaches are presented using the verification system PVS. One approach declares process terms as an uninterpreted type and specifies equality on terms by axioms. This is convenient for concrete applications where the rewrite mechanisms of PVS can be exploited. For the verification of theoretical results, often induction principles are needed. They are provided by the second approach where process terms are defined as an abstract datatype with a separate equivalence relation.

1 Introduction

We investigate the possibilities of obtaining mechanical support for equational reasoning in process algebra. In particular, we consider ACP-style process algebras [2, 3], where processes are represented by terms constructed from atoms (denoting atomic actions) and operators such as choice (non-determinism), sequential composition, and parallel composition. Axioms specify which process terms are considered to be equal.

The idea is to apply equational reasoning to processes, similar to normal arithmetic. This reasoning is often very tedious and error-prone, and it is difficult to check all details manually. Especially concurrency, which is usually unfolded into a sequential term representing all interleavings, might generate large and complex terms. Hence, the quest for convenient proof support for process algebra. We investigate two aspects:

– Mechanical support for the verification of concrete applications. The aim is usually to verify that an implementation satisfies a specification. Both are expressed in process algebra, where the implementation is more detailed with additional (internal) actions. The goal is to show that the specification equals the implementation after the abstraction from internal actions. The proof proceeds by rewriting the implementation using the axioms until the specification is obtained.
– Mechanical support for the proof of theoretical properties of a process algebra. A common proof technique is based on so-called elimination theorems. Such a theorem states that any closed process term in a given process algebra can be rewritten into a basic term, i.e. a term consisting of only atoms, choices, and atom-prefixes (restricted sequential composition). Thus a property for general process terms can be reduced into one for basic terms, which can then be proved by induction on the structure or the length of basic terms.

Since our goal is to reason about recursive, possibly infinite, processes and to verify not only concrete applications, but also general theoretical results, we do not aim at completely automatic verification. In this paper, we investigate how process algebra can be incorporated in the framework of the tool PVS (Prototype Verification System) [16]. Properties can be proved in PVS by means of an interactive proof checker. This means that the user applies proof commands to simplify the goal that must be proven, until it can be proved automatically by the powerful decision procedures of the tool.

We experiment with two different definitions of process algebra in the specification language of PVS, a typed higher-order logic. One possibility is to define process terms by means of the abstract-datatype mechanism of PVS which generates, among others, a useful induction scheme for the datatype, allowing induction on the structure of terms. As an alternative, we investigate how the rewriting mechanisms of PVS can be exploited for equational reasoning. Since process algebra, as a method for specifying and verifying complex systems, is still under development, many different algebras already exist and others are still being designed. Therefore, the goal is to create a flexible framework in PVS that allows experiments with tool support for customized process algebras.

Related Work. A lot of effort has been devoted to the development of dedicated tools for process algebra. For PSF [13], an extension of ACP with abstract datatypes, tools are available that mainly support specification and simulation. PAM [12] is a related tool which provides flexible language support. Another class of dedicated tools aims at automatic verification, including bisimulation and model checkers. An example is the Concurrency Factory [8].

More related to our work is research on the use of general purpose proof checkers. E.g., tool support for CCS and CSP has been obtained using HOL [6, 7, 15]. This theorem prover has also been used to get mechanized support for reasoning with the π-calculus [14]. For μCRL, an ACP-like language with data structures, both Coq [5, 11] and PVS [10] have been investigated. In [5] pure algebraic reasoning is used, whereas the work described in [10, 11] combines algebraic and assertional reasoning.

Most of the research mentioned above aims at concrete applications. The only support for the verification of theoretical issues concerns the soundness proof of algebraic axioms, based on a specific semantic model [6, 14, 15]. Whereas this often concerns theory about the underlying model, we are more interested in the verification of theoretical results on the axiomatic level, without relying on any underlying model.

Also different is that we explicitly study the choices that can be made when incorporating process algebra in a general purpose proof checker. In that respect, our work is probably most related to research on tool support for a CSP-like algebra by means of HOL [9]. In fact, they investigate similar approaches as we do, although they only consider small concrete examples. New in our paper is, besides the verification of non-trivial examples, that we additionally show how to obtain proof support for the development of ACP-style theory on the axiomatic level.

Overview. In Section 2, we investigate two alternatives for the modeling of process algebra in PVS. An approach where process terms are defined as an abstract datatype, with a separate equivalence relation on terms, is presented in Section 3. It is used to prove a number of theoretical results, using induction schemes provided by PVS. Section 4 describes an alternative approach where process terms are defined as an uninterpreted type, allowing convenient rewriting of concrete process terms. Concluding remarks can be found in Section 5.

2 Modeling Process Algebra in PVS

We discuss two approaches to defining process algebra in PVS. First, in Section 2.1, we briefly introduce the process-algebraic framework considered in this paper. A straightforward formulation in PVS, using uninterpreted types plus equality, is presented in Section 2.2. An approach where terms are defined as an abstract datatype is described in Section 2.3.

2.1 Process Algebra

To illustrate the main concepts, we consider theory PA (Process Algebra), as defined in [2, 3]. This theory is presented in Table 1, where parameter A represents the set of atoms. The first entry of this table specifies the sorts; P is the sort of all process terms.

The second entry lists the standard algebraic operators; choice, denoted $+$, sequential composition, denoted $\cdot$, parallel composition or merge, denoted $\|$, and an auxiliary operator called the left merge, denoted $\llfloor$, which is used to axiomatize the merge. Intuitively, the left merge corresponds to parallel execution, with the restriction that the left process executes the first action.

The third entry of Table 1 contains the axioms. For instance, Axiom $A4$ specifies right-distributivity of sequential composition over choice. The absence of left-distributivity implies that processes with different moments of choice are distinguished. The axioms define an equivalence relation on processes. A model of these axioms, thereby showing their consistency, consists of equivalence classes of closed terms (i.e. terms without variables) as processes, with bisimulation as the equivalence relation. Note, however, that this is only one possible model. A strong point of axiomatic reasoning is that it is model independent.

Standard for equational specifications are general substitution and context rules which express that a process can be replaced by an equivalent term in any context, i.e., inside any term.

PA(A)			
$P;\ \mathsf{A} \subseteq P$			
$\_ + \_,\ \_ \cdot \_,\ \_ \| \_,\ \_ \llfloor \_ \ :\ P \times P \to P$			
$a:\ \mathsf{A};\ x, y, z:\ P;$			
$x + y = y + x$	$A1$	$x \| y = x \llfloor y + y \llfloor x$	$M1$
$(x + y) + z = x + (y + z)$	$A2$	$a \llfloor x = a \cdot x$	$M2$
$x + x = x$	$A3$	$a \cdot x \llfloor y = a \cdot (x \| y)$	$M3$
$(x + y) \cdot z = x \cdot z + y \cdot z$	$A4$	$(x + y) \llfloor z = x \llfloor z + y \llfloor z$	$M4$
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$A5$		

Table 1. The process algebra PA.

2.2 Using Uninterpreted Types plus Equality

In PVS theory `PArew`, we model process algebra PA with the intention to exploit the rewriting mechanisms of PVS. Theory `PArew` is parameterized by the type `Atoms`. Process terms are just defined as some non-empty uninterpreted type, assuming a function `trm` which maps atoms into terms. This function is defined as a *conversion* in PVS, which means that it need not be mentioned explicitly.

```
PArew [Atoms: NONEMPTY_TYPE]: THEORY
BEGIN
Terms : NONEMPTY_TYPE
trm   : [Atoms -> Terms]
CONVERSION trm
```

Next we define the operators as functions in the language of PVS and axiomatize equality on terms, using the built-in equality on uninterpreted types. Frequently, atoms are interpreted as terms using conversion `trm`. E.g., `a o x` is interpreted as `trm(a) o x`. Moreover, note that `o` binds stronger than `//` which binds stronger than `+`.

```
+, o, //, lmrg : [Terms,Terms -> Terms]
a         : VAR Atoms
x, y, z : VAR Terms
A1 : AXIOM          x + y = y + x
A2 : AXIOM    (x + y) + z = x + (y + z)
A3 : AXIOM          x + x = x
A4 : AXIOM    (x + y) o z = x o z + y o z
A5 : AXIOM    (x o y) o z = x o (y o z)
M1 : AXIOM         x // y = lmrg(x,y) + lmrg(y,x)
M2 : AXIOM      lmrg(a,x) = a o x
M3 : AXIOM  lmrg(a o x,y) = a o (x // y)
M4 : AXIOM  lmrg(x + y,z) = lmrg(x,z) + lmrg(y,z)
END PArew
```

In general, one should be careful with axioms in PVS, because they might introduce inconsistencies. However, as mentioned in Section 2.1, there are several models satisfying the above axioms, showing that they are consistent. For the time being, we did not formalize a model in PVS, since our main interest concerns proof support for ACP-style axiomatic reasoning. When using PVS for a customized process algebra, its consistency must of course be shown by providing a model.

As a simple application of this theory, we present theory `PArewex` which imports `PArew`. The theorem called `expand` shows the equivalence of a parallel process and a sequential term, representing all interleavings. This theorem can be proved automatically in PVS after installing automatic rewrites on all axioms except `A1`.

```
PArewex  : THEORY
BEGIN
Atoms : TYPE = {a,b,c,d}
IMPORTING PArew[Atoms]
expand : THEOREM (a+b) o (a+b) // (c+d) =
      a o (a o (c + d) + b o (c + d) + (c o (a + b) + d o (a + b))) +
      b o (a o (c + d) + b o (c + d) + (c o (a + b) + d o (a + b))) +
      c o (a o (a + b) + b o (a + b)) +
      d o (a o (a + b) + b o (a + b))
END PArewex
```

In Section 4, we illustrate this approach by a more complex process algebra and a non-trivial example. However, this framework is not suitable for proving theoretical results, based on inductive proofs.

2.3 Defining Process-Algebra Terms as an Abstract Datatype

Proofs about properties of process algebra often use induction on the structure of terms. Since PVS generates such induction schemes for abstract datatypes, it seems convenient to model process terms as an abstract datatype. Hence we present an approach in which the terms of PA are represented as an abstract datatype with type `Atoms` as a parameter. The datatype below contains five so-called *constructors*: `atm` to turn atoms into terms, and four operators `o`, `+`, `//`, and `lmrg` for, resp., sequential composition, choice, merge and left merge.

```
PA_terms [Atoms: TYPE] : DATATYPE
BEGIN atm(at: Atoms)              : atom?
      o(sq1, sq2: PA_terms)       : seq?
      +(ch1, ch2: PA_terms)       : choice?
      //(mrg1, mrg2: PA_terms)    : merge?
      lmrg(lmrg1, lmrg2: PA_terms) : lmerge?
END PA_terms
```

When type checking this datatype definition, the PVS system generates a new file which contains a large number of useful definitions and properties of the datatype. E.g., a subterm relation `x << y` is defined, with an axiom to express that it is well-founded.

```
PA_terms_well_founded: AXIOM well_founded?[PA_terms](<<);
```

Moreover, an induction scheme is generated, expressing that a property `p` on terms can be proved by showing that it holds for all atoms and by proving that it holds for the other operators if the subterms already satisfy `p`.

Defining an Equivalence Relation on Terms. Observe that for terms that are defined as an abstract datatype, equality has a fixed meaning in PVS, namely syntactic equality. Hence, equality cannot be used to express equivalence of process terms, as we did in Section 2.2. Therefore, we define in PVS theory `PA` a separate equivalence relation, denoted `==`, on PA terms, using a pre-defined predicate `equivalence?` which implies that the relation is reflexive, symmetric, and transitive. As before, this relation is specified by the axioms of PA.

```
PA[Atoms: NONEMPTY_TYPE]: THEORY
BEGIN
IMPORTING PA_terms[Atoms]
== : (equivalence?[PA_terms])
a, b, c       : VAR Atoms
v, w, x, y, z : VAR PA_terms

A1: AXIOM          x + y == y + x
...
M4: AXIOM lmrg(x + y, z) == lmrg(x, z) + lmrg(y, z)
```

Henceforth, we omit variable declarations if they have been presented in earlier theories.

Standard for equational reasoning is that equivalent terms can be substituted by one another in contexts. Unfortunately, in the current framework, this has to be expressed explicitly as follows.

```
ch_l:  AXIOM x == z IMPLIES  x + y == z + y
...
mrg_l: AXIOM x == z IMPLIES x // y == z // y
```

Due to the possibility to do inductive proofs, the approach of the current subsection provides a more powerful framework than the one of Section 2.2. Therefore, we first study the applicability of the approach with abstract datatypes in the next section.

3 Abstract Data Types plus Equivalence Relation

The most interesting aspect of the use of abstract datatypes for process terms is that it allows inductive proofs. As mentioned in the introduction, inductive proofs are often used in a proof technique based on basic terms and an elimination theorem. Section 3.1 briefly explains this proof technique. Section 3.2 contains a formulation of basic terms in PVS. In Section 3.3, we prove in PVS that each closed PA term can be translated into an equivalent basic term. This translation is used in Section 3.4 to prove properties about the alphabet of a process term and in Section 3.5 to show associativity of the merge operator.

3.1 Basic Terms and Elimination

For a convenient treatment of realistic examples, most process algebras contain a large number of operators and axioms. This, however, complicates the proof of general properties about the algebra. Hence, it is extremely useful if one can show that many operators can be eliminated and any term can be reduced to an equivalent term with only a few basic operators. In the framework of [2, 3], this leads to the concept of *basic terms*.

Definition (Basic terms). The set of basic terms is inductively defined as follows. The atoms A are contained in the set of basic terms. Furthermore, for any $a \in \mathsf{A}$ and basic terms s, t, also $a \cdot t$ and $s + t$ are basic terms. No other terms are basic terms.

It can be shown that any closed PA term can be translated into an equivalent basic term.

Theorem (Elimination). For any closed PA term p, there exists a basic term t, such that $p = t$ can be derived from the axioms of PA.

A standard proof technique for a property on process terms, is to reduce it by the elimination theorem to a property on basic terms which is then proved by induction on the structure of basic terms (see Section 3.4) or the length of basic terms (see Section 3.5). This axiomatic reasoning is model independent and hence the property holds in any model based on closed terms as processes.

3.2 Defining Basic Terms in PVS

To define basic terms, we extend theory `PA` of Section 2.3 with a predicate `basic?` on the abstract datatype `PA_terms`. This predicate is defined recursively on the structure of PA terms. In PVS, this requires a so-called *measure* function which should be well-founded and should decrease with every recursive call. In general, type checking in PVS need not be decidable; it might generate so-called Type Check Conditions (TCCs) which are proof obligations that have to be fulfilled for type correctness. For recursive definitions, TCCs concerning the correctness of the measure function are generated.

```
basic?(x) : RECURSIVE bool =
              CASES x OF
                atm(a)     : TRUE,
                o(y, z)    : atom?(y) AND basic?(z),
                +(y, z)    : basic?(y) AND basic?(z),
                //(y, z)   : FALSE,
                lmrg(y, z): FALSE
              ENDCASES
            MEASURE x BY <<;
basic_terms : TYPE = {x | basic?(x)}
```

The recursive definition above leads to several TCCs, including one requiring that the subterm relation << is well-founded. This follows immediately from the corresponding axiom mentioned in Section 2.3. The other TCCs, requiring that the recursive calls are applied to subterms of argument x, are trivial and can be proved automatically.

Note that we have not defined basic terms as a separate abstract datatype; by defining it as a predicate on datatype PA_terms, we obtain the desired subtype relation between basic terms and process terms. This subtype relation is crucial for proofs based on the elimination theorem, as shown by the applications in the remainder of this section.

3.3 Translating PA Terms to Basic Terms

In theory PA2Basic we prove the elimination theorem. Note that the datatype PA_terms does not contain variables, which means that it specifies closed terms. We define a translation function pa2b which maps PA terms into basic terms. This definition is recursive and uses relations < and <= on terms that are presented below.

```
PA2Basic[Atoms: NONEMPTY_TYPE]: THEORY
...
pa2b(x): RECURSIVE {b: basic_terms | b <= x} =
      CASES x OF
        atm(a)     : atm(a),
        o(y, z)    : CASES pa2b(y) OF
                       atm(a) : atm(a) o pa2b(z),
                       o(v, w): v o pa2b(w o z),
                      +(v, w): pa2b(v o z) + pa2b(w o z)
                     ENDCASES,
        +(y, z)    : pa2b(y) + pa2b(z),
        //(y, z)   : pa2b(lmrg(y, z)) + pa2b(lmrg(z, y)),
        lmrg(y, z): CASES pa2b(y) OF
                       atm(a) : atm(a) o pa2b(z),
                       o(v, w): v o pa2b(w // z),
                       +(v, w): pa2b(lmrg(v, z)) + pa2b(lmrg(w, z))
                     ENDCASES
      ENDCASES
         MEASURE x BY <;
```

This definition generates 26 TCCs to show, for instance, that recursive calls are applied to terms that are smaller than argument x, according to the relation <, and to show that the result of the function is a basic term not greater than the argument, according to <=.

The main problem was to find definitions for the relations `<` and `<=` such that all TCCs could be proved. For instance, some obvious relations based on the length of terms (the number of symbols) are not correct, since we have to show, for example, that `pa2b(lmrg(y, z)) + pa2b(lmrg(z, y)) <= y // z`.

The solution is based on a weight function on PA terms mentioned in [1]. It uses the exponentiation function `expt`, which is already available in PVS.

```
weight(x): RECURSIVE {n: nat | n >= 2} =
             CASES x OF
               atm(a)     : 2,
               o(y, z)    : weight(y) * weight(z) + weight(y),
               +(y, z)    : weight(y) + weight(z) + 1,
               //(y, z)   : expt(2, weight(y) + weight(z) + 2),
               lmrg(y, z): expt(2, weight(y) + weight(z))
             ENDCASES
           MEASURE x BY <<;
<(x, y)  : bool = weight(x) < weight(y)
<=(x, y) : bool = x < y OR x = y
```

The elimination theorem, `pa2b_eq`, expresses that the result of translation `pa2b` is equivalent to its argument. The proof is rather tedious and uses induction over the structure of argument `x` as provided by the induction mechanism generated by `PA_terms`.

```
pa2b_eq: THEOREM pa2b(x) == x
END PA2Basic
```

Our proof of this elimination theorem is constructive in the sense that it provides a concrete transformation from PA terms to basic terms. This in contrast with the literature on process algebra where the proofs usually rely on term-rewriting theory [2, 3].

3.4 Using Elimination and Structural Induction on Basic Terms

As a simple application of the elimination theorem, we define the alphabet of PA terms by means of three axioms that specify the alphabet of basic terms. Additionally, Axiom `AB4` specifies that equivalent terms have the same alphabet.

```
Alpha [Atoms: NONEMPTY_TYPE]: THEORY
BEGIN
IMPORTING PA2Basic [Atoms]
btx, bty, btz  : VAR basic_terms
alpha : [PA_terms -> setof[Atoms]]  % alphabet
AB1: AXIOM           alpha(atm(a)) = singleton(a)
AB2: AXIOM       alpha(atm(a) o x) = add(a,alpha(x))
AB3: AXIOM            alpha(x + y) = union(alpha(x),alpha(y))
AB4: AXIOM x == y IMPLIES alpha(x) = alpha(y)
```

We show that this implies the expected property for the alphabet of a general sequential composition, as stated in theorem `AB2pa`. Using theorem `pa2b_eq`, it is sufficient to prove the property for basic terms, as expressed by lemma `AB2b`.

```
AB2b  : LEMMA   alpha(btx o bty) = union(alpha(btx),alpha(bty))
AB2pa : THEOREM     alpha(x o y) = union(alpha(x),alpha(y))
END Alpha
```

Lemma `AB2b` has been proved by induction on `btx`; this gives induction on the structure of `PA_terms`, but since `basic?(btx)` holds, the cases for non-basic terms can be discharged trivially. Hence, this boils down to induction on the structure of basic terms.

3.5 Using Elimination and Induction on the Length of Basic Terms

General properties of the merge and the left merge are often useful in verifications. For process algebra PA these properties can be proved, in a model-independent way, by means of the elimination theorem. For a process algebra with recursion this is not always possible, and then they are introduced as axioms, called the *axioms of standard concurrency* [2, 3]. In this section, we concentrate on associativity of the merge, called ASC6. The proof uses commutativity of the merge, called ASC2, and a property of the left merge, ASC4. The other axioms of standard concurrency deal with communication and are omitted here. Property `ASC2` is proved easily using `M1` and `A1`.

```
PAsc [Atoms: NONEMPTY_TYPE]: THEORY
BEGIN
IMPORTING PA2Basic[Atoms]
ASC2 : THEOREM  x // y == y // x
```

To prove the other two properties, elimination theorem `pa2b_eq` is used to reduce these properties to basic terms, as expressed by `ASC4b` and `ASC6b`. By lemma `ASC46b`, these two lemmas are proved simultaneously by strong natural induction on the sum of the lengths of the basic terms. The proof of `ASC46b` also uses case analysis on the structure of basic terms, illustrating the importance of the reduction to basic terms.

```
length(x) : RECURSIVE posnat =
              CASES x OF
                atm(a)    : 1,
                o(x,y)    : length(x) + length(y),
                ...       % similar for +, //, lmrg
              ENDCASES
            MEASURE x by <<
n  : VAR nat
ASC46b : LEMMA
   (FORALL btx,bty,btz: n = length(btx) + length(bty) + length(btz)
              IMPLIES lmrg(lmrg(btx,bty),btz) == lmrg(btx,bty//btz) )
   AND
   (FORALL btx,bty,btz: n = length(btx) + length(bty) + length(btz)
              IMPLIES (btx//bty)//btz == btx//(bty//btz) )

ASC4b : LEMMA lmrg(lmrg(btx,bty),btz) == lmrg(btx,bty//btz)
ASC6b : LEMMA         (btx//bty)//btz == btx//(bty//btz)
ASC4  : THEOREM     lmrg(lmrg(x,y),z) == lmrg(x,y // z)
ASC6  : THEOREM             (x//y)//z == x//(y//z)
END PAsc
```

The proofs of theorems `ASC4` and `ASC6` use lemma `pa2b_eq` to replace `x`, `y`, and `z` by equivalent basic terms. The proofs are completed using symmetry, transitivity, and a few properties of == about substitution in a context.

Observe that rewriting is cumbersome in the current approach because symmetry, transitivity, rewriting in contexts, etc., all have to be performed explicitly. Although this can be solved to some extent by defining a strategy in PVS that combines these commands, it would be more convenient if the user could define its own congruence relation, such as ==, and obtain the desired rewriting. The main conclusion of this section is that the PVS facilities for abstract datatypes and subtyping are useful to prove non-trivial theorems in process-algebra theory with a reasonable amount of effort.

4 Verifying Applications Using Equational Reasoning

Since rewriting turned out to be tedious in the proofs of the previous section, we elaborate in this section on the approach of Section 2.2 where terms are defined as an uninterpreted type with axioms that specify equality on terms. In order to experiment with this approach on some more complicated applications, we axiomatize $ACP^{\tau*}$: Algebra of Communicating Processes (ACP) with abstraction [3] and binary Kleene star [4]. The formal framework is defined in Section 4.1 and applied to the verification of an Alternating-Bit Protocol (ABP) in Section 4.2. This protocol often serves as a benchmark for verifications in process algebra [3, 5, 11].

4.1 Defining ACP by Uninterpreted Terms and Equality

Similar to PA, process algebra $ACP^{\tau*}$ contains atoms and operators for sequential composition, choice, merge and left merge. In addition, there are two special atoms δ, indicating deadlock or unsuccessful termination, and τ, representing the silent (internal) step. The merge in $ACP^{\tau*}$ is slightly different; besides interleaving the atoms of the two processes, represented by the left merge, it is now also possible to have a *synchronous communication*, represented by a communication merge "|". This communication merge is defined by means of a *communication function* γ which defines, for a particular application, the result of the communication for each pair of atoms. A result δ indicates that the atoms cannot synchronize. The axioms of $ACP^{\tau*}$ axiomatize rooted branching bisimulation, which means that processes with the same external behavior, but possibly different internal actions, are considered to be equal. This equivalence is particularly suitable to verify implementations versus specifications, as explained in the next subsection.

Theory `ACPtbks` implements $ACP^{\tau*}$ in PVS. It has a communication function as a parameter and contains explicit assumptions about its properties. If a theory imports `ACPtbks` with a particular function, TCCs are generated to show that the assumptions are fulfilled.

```
ACPtbks [Atoms: NONEMPTY_TYPE, delta: Atoms, tau: Atoms,
         gamma: [Atoms, Atoms -> Atoms] ]: THEORY
BEGIN
ASSUMING
   C1 : ASSUMPTION          gamma(a,b) = gamma(b,a)
   C2 : ASSUMPTION gamma(gamma(a,b),c) = gamma(a,gamma(b,c))
   C3 : ASSUMPTION      gamma(a,delta) = delta
   C4 : ASSUMPTION        gamma(a,tau) = delta
ENDASSUMING
```

The definition of `Terms`, conversion `trm`, operators `+, o, //, lmrg`, and axioms `A1` through `A5` are exactly the same as in theory `PArew` of Section 2.2.

New are axioms for `delta` and `tau`, the definition of the communication merge `/`, and a changed list of axioms for concurrency. Note that `B1` and `B2` express that `tau` is not observable and can be removed, provided all options present before executing the silent action are present after executing it. Not shown are `CM2 - CM4`, which are equal to `M2 - M4`, some axioms for `/`, and the axioms of standard concurrency [2, 3].

```
A6 : AXIOM               x + delta = x
A7 : AXIOM               delta o x = delta
B1 : AXIOM                 x o tau = x
B2 : AXIOM x o (tau o (y + z) + y) = x o (y + z)

/: [Terms,Terms -> Terms]
CF  : AXIOM            a / b = gamma(a,b)
CM1 : AXIOM            x // y = lmrg(x,y) + lmrg(y,x) + (x / y)
...
CM9 : AXIOM       x / (y + z) = (x / y) + (x / z)
```

The encapsulation operator `enc` maps atoms of a set `H` to `delta`. It can be used to enforce that certain atoms communicate; they cannot occur in isolation. Not shown here are similar axioms that specify the abstraction operator `abs` which hides (internal) atoms of a set by mapping them to the silent action `tau`.

```
enc, abs  : [setof[Atoms], Terms -> Terms]
H   : VAR setof[Atoms]
D1  : AXIOM NOT member(a,H) IMPLIES enc(H,a) = a
D2  : AXIOM     member(a,H) IMPLIES enc(H,a) = delta
D3  : AXIOM                    enc(H, x + y) = enc(H,x) + enc(H,y)
D4  : AXIOM                    enc(H, x o y) = enc(H,x) o enc(H,y)
```

The binary Kleene star represents an iteration; `x * y` denotes the process that can repeatedly behave as the body `x`, but it can non-deterministically stop the repetition and decide to behave as `y`. We only show the axioms that are needed in the next section.

```
*    : [Terms,Terms -> Terms]
BKS1 : AXIOM                       x * y = (x o (x * y)) + y
BKS4 : AXIOM               enc(H, x * y) = enc(H,x) * enc(H,y)
```

The Fair Iteration Rule, `FIR`, excludes an infinite sequence of `tau` atoms if there is an alternative. The Recursive Specification Principle for the binary Kleene star, `RSPbks`, specifies the solution of a particular form of guarded recursive equations. A term `x` is guarded, denoted `guard?(x)`, if it cannot terminate successfully without performing at least one visible action.

```
FIR    : AXIOM tau * x = x + (tau o x)
RSPbks : AXIOM guard?(y) AND x = (y o x) + z IMPLIES x = y * z
END ACPtbks
```

4.2 Verification of an Alternating-Bit Protocol

To experiment with the framework of the previous subsection, we consider a version of the ABP with iteration and fairness. The verification of this protocol follows a standard approach which is the basis for any ACP-style verification; after the introduction of a few basic primitives, first the required service is specified. Next the implementation of the protocol is specified and we show that, after encapsulation and abstraction, it is equivalent to the specification.

For the ABP, we need message passing with bits. Therefore, atoms are structured as an abstract datatype. Besides `delta` and `tau`, we have input, output, send, receive, and communication atoms. Input and output atoms represent the communication with the environment of the protocol, i.e., they represent its external interface, whereas send and receive are internal atoms that synchronize to a communication atom.

```
ABP_Atoms [Messages : TYPE, Bits : TYPE] : DATATYPE
  BEGIN delta                         : delta?
        tau                           : tau?
        inp(im: Messages)             : imsg?
        outp(om: Messages)            : omsg?
        send(sm: Messages, sb: Bits) : send?
        rec(rm: Messages, rb: Bits)  : rec?
        comm(cm: Messages, cb: Bits) : comm?
  END ABP_Atoms
```

This general structure is used in theory ABP with simple messages representing data d and acknowledgments a. Bits are represented by t (true) and f (false). Alternation of bits is defined by function alt. The hand-shake communication mechanism is defined by function gamma. It expresses that a send and a receive should be combined into a communication. Observe that importing ACPtbks leads to four TCCs corresponding to the assumptions on gamma.

```
ABP : THEORY
BEGIN
Messages : TYPE = {d,a}
Bits     : TYPE = {t,f}
IMPORTING ABP_Atoms[Messages,Bits]
m, m0   : VAR Messages
b, b0   : VAR Bits
e, f, g : VAR ABP_Atoms
alt(b)     : Bits = CASES b OF t : f, f : t ENDCASES
gamma(e,f) : ABP_Atoms =
  CASES e OF
    send(m,b): CASES f OF
       rec(m0,b0): IF m0=m AND b0=b THEN comm(m,b) ELSE delta ENDIF
       ELSE delta ENDCASES,
    rec(m,b) : ...   % similarly
    ELSE delta
  ENDCASES
IMPORTING ACPtbks[ABP_Atoms,delta,tau,gamma]
```

The aim is to verify an ABP according to specification ABP_spec which expresses that it should behave as a one-place buffer, copying data on its input port to its output port. Note that x*delta denotes a non-terminating iteration, repeating body x forever (see axioms BKS1 and A6).

```
ABP_spec: Terms = (inp(d) o outp(d))*delta
```

This specification is implemented by means of a sender and a receiver. For simplicity, we do not model the communication channels between them, but assume they communicate directly; channel failures are modeled in the behavior of the receiver.

The sender S alternates between S(t) and S(f), where S(b) gets a data item, sends it with bit b, and next repeatedly receives an erroneous acknowledgment (expressed by SE(b)) until it gets a correct one (expressed by SN(b)).

```
SE(b) : Terms = rec(a,alt(b)) o send(d,b)       % error part sender
SN(b) : Terms = rec(a,b)                        % normal part sender
S(b)  : Terms = inp(d) o send(d,b) o ( SE(b) * SN(b) )
S     : Terms = (S(t) o S(f))*delta
```

The receiver has a similar structure; its error part, denoted by term RE(b), models in an abstract way the possibility that messages might be corrupted by the channel and the receiver sends an acknowledgment with the wrong bit. Note that this error part can be repeated indefinitely. However, assuming fairness, only a finite number of subsequent errors can occur.

```
RE(b) : Terms = send(a,alt(b)) o rec(d,b)       % error part receiver
RN(b) : Terms = outp(d) o send(a,b)             % normal part receiver
R(b)  : Terms = rec(d,b) o ( RE(b) * RN(b) )
R     : Terms = (R(t) o R(f))*delta
```

Further, we define a set H which is used to encapsulate isolated send and receive atoms, and a set I which is used to abstract from communication events.

```
H: setof[ABP_Atoms] = { e | send?(e) OR rec?(e) }
I: setof[ABP_Atoms] = { e | comm?(e) }
```

Then, the aim is to prove that the parallel composition of the sender and the receiver, S // R, encapsulating the send and receive atoms and abstracting from the communication atoms, equals the specification of the Alternating-Bit Protocol;

```
abs(I,enc(H,S // R)) = ABP_spec.
```

First, we show that enc(H,S//R) = X, where X is an auxiliary term defined by:

```
XE(b) : Terms = comm(a,alt(b)) o comm(d,b)      % error part protocol
XN(b) : Terms = outp(d) o comm(a,b)             % normal part protocol
X(b)  : Terms = inp(d) o comm(d,b) o ( XE(b) * XN(b) )
X     : Terms = (X(t) o X(f))*delta
```

We start with unfolding the iterations inside S // R using BKS1, which makes the choice between the normal and error parts in the body of each component explicit. Next, we prove that the body of the protocol equals the body of X. This is far from trivial, but the proof in PVS is rather straightforward. First, we install a large number of axioms and some useful lemmas as automatic rewrite rules (from left to right). This includes A2, A5, A6, A7, CF, CM2 through CM9, D3, D4, and BKS4. Then, we repeatedly rewrite explicitly using CM1, expand the definition of gamma and apply the automatic rewrites. This proof shows the main advantage of using equality over a user-defined congruence; substitution in contexts, transitivity, etc., are all implicitly incorporated in the rewriting mechanism of PVS. Using a similar rewriting, we can then prove that one iteration of S in parallel with R corresponds to one iteration of X.

```
once_rep : LEMMA enc(H,S // R) = (X(t) o X(f)) o enc(H,S // R)
```

Recursion axiom RSPbks (with delta instead of z) and Axiom A6 then lead to

```
cbhv : LEMMA enc(H,S // R) = X
```

Finally, using the properties of abstraction and fairness principle FIR, we obtain

```
ABP_eq_spec : THEOREM abs(I,enc(H,S // R)) = ABP_spec
END ABP
```

Comparing our proof using PVS with a manual proof, one can observe that the main proof steps are the same. In a manual proof, however, usually not all intermediate steps are written down, whereas a tool such as PVS requires a detailed check of all steps. Fortunately, these tedious steps can be automated to a large extent, using the powerful rewrite capabilities of PVS. This leads to a higher degree of automation than a related verification in the proof checker Coq [5]. The authors of [5] explicitly mention that rewriting is not so easy in Coq.

5 Concluding Remarks

Two approaches have been presented to formulate ACP-like process algebras [2, 3] in the language of PVS. Each approach has been validated by applying it to non-trivial examples.

Process terms as an uninterpreted type. Equality on terms is specified by means of axioms that can be used as automatic rewrite rules by the PVS proof checker. In this framework, we have formalized an algebra of communicating processes with abstraction and the binary Kleene star as a limited form of recursion. A disadvantage of this approach is the lack of induction principles, which are essential for proofs of theoretical results about process algebra.

Process terms as abstract datatype. In this approach, an additional equivalence relation on terms is introduced and axiomatized. By using the abstract-datatype mechanism of PVS, we obtain convenient induction principles. A disadvantage of this framework is that substitution in contexts has to be formalized explicitly and rewriting by means of the equivalence relation is inconvenient.

Conclusion. The main conclusion is that mechanical support for process algebra by means of PVS is feasible, both for theory development and for concrete applications. New in our paper is that we have obtained suitable tool support for the proof of theoretical properties of ACP-style algebras. We have proved an elimination theorem which can be used to prove a property of an algebra by reducing it to a property formulated in basic terms. Besides its use for applications such as verified in this paper, the elimination theorem also plays a role in completeness proofs for specific models [2, 3].

Unfortunately, the ideal framework for theory development differs from the ideal framework for concrete applications. It would be a major improvement if the two approaches can be combined, allowing inductive proofs and convenient term rewriting. Ideally, this could be achieved by extending the PVS system with the facility to perform rewriting on user-defined congruence relations. An alternative is to define powerful proof strategies that incorporate general rewrite patterns for congruences.

As mentioned in the introduction, essentially the same approaches as the ones we studied here are investigated in [9] where a CSP-like process algebra is embedded in HOL. The conclusions of [9] about rewriting and equational reasoning are similar to ours. As a result, the authors express a slight preference for the approach with uninterpreted types. However, in [9] only small concrete examples have been studied and no theoretical results have been derived. Our work shows that when one is interested in theory development for ACP-style process algebras, the approach based on abstract datatypes is the only one feasible. Also note that we heavily use the subtyping mechanism of PVS (to define basic terms) and dependent types, features which are not supported by the HOL system.

An advantage of the use of a general purpose verification tool such as PVS, above a dedicated process-algebra tool, is the possibility to get insight in the desired tool support for various fields of application and different process algebras in a short amount of time. Using the large number of predefined theories and libraries, it is easy to study extensions and variations of the framework. As an alternative to PVS, it would be interesting to experiment with the generic theorem prover Isabelle [17], since it allows rewriting with user-defined congruence relations and ordered rewriting (allowing, e.g., rewriting using a commutativity axiom).

Acknowledgments. We would like to thank Jaco van de Pol and Jos Baeten for their comments on a draft version of this paper.

References

1. G.J. Akkerman and J.C.M. Baeten. Term rewriting analysis in process algebra. *CWI Quarterly*, 4(4):257–267, 1991.
2. J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, *Semantic Modelling*, pages 149–268. Oxford University Press, Oxford, UK, 1995.
3. J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Prentice-Hall, 1990.
4. J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration and nesting. *The Computer Journal*, 37(4):241–258, 1994.
5. M.A. Bezem, R.N. Bol, and J.F. Groote. Formalizing process algebraic verifications in the calculus of constructions. *Formal Aspects of Computing*, 9(1):1–48, 1997.
6. A. Camilleri. A Higher Order Logic mechanization of the CSP failure-divergence semantics. In *IV Higher Order Workshop*, pages 123–150. Workshops in Computing, Springer-Verlag, 1991.
7. A. Camilleri, P. Inverardi, and M. Nesi. Combining interaction and automation in process algebra verification. In *TAPSOFT'91*, pages 283–296. LNCS 494, Springer-Verlag, 1991.
8. R. Cleaveland, J. Gada, P. Lewis, S. Smolka, O. Sokolsky, and S. Zhang. The Concurrency Factory – practical tools for specification, simulation, verification, and implementation. In *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, 1994.
9. R. Groenboom, C. Hendriks, I. Polak, J. Terlouw, and J.T. Udding. Algebraic proof assistants in HOL. In *Math. of Program Construction*, pages 304–321. LNCS 947, Springer-Verlag, 1995.
10. J.F. Groote, F. Monin, and J. Springintveld. A computer checked algebraic verification of a distributed summation algorithm. Computing Science Report 97/14, Eindhoven University of Technology, The Netherlands, 1997.
11. H. Korver and A. Sellink. On automating process algebra proofs. In *Proc. Symp. on Computer and Information Sciences, ISCIS XI*, volume II, pages 815–826, 1996.
12. H. Lin. PAM: A process algebra manipulator. In *Proc. Third Workshop on Computer Aided Verification*, pages 136–146. LNCS 575, Springer-Verlag, 1991.
13. S. Mauw and G.J. Veltink. A proof assistant for PSF. In *Proc. Third Workshop on Computer Aided Verification*, pages 158–168. LNCS 575, Springer-Verlag, 1991.
14. T.F. Melham. A mechanized theory of the π-calculus in HOL. Technical Report 244, Computer Laboratory, University of Cambridge, 1992.
15. M. Nesi. Value-passing CCS in HOL. In *Proc. 6th Workshop on Higher Order Logic Theorem Proving and Applications*, pages 352–365. LNCS 780, Springer-Verlag, 1993.
16. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Software Eng.*, 21(2):107–125, 1995.
17. L.C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828, Springer-Verlag, 1994.