Formalising LPOs and Invariants in Coq

Henri Korver
Alex Sellink
Department of Philosophy, Utrecht University
P.O. Box 80.126, 3508 TC Utrecht, The Netherlands
Email: {korver,alex}@phil.ruu.nl

Abstract

In the setting of μCRL, the notions of 'linear process operator (LPO)' and 'invariant' are implemented in Coq, which is a a proof development tool based on type theory. As a first experiment we have computer-checked a general property of a binary search program in the new framework.

1 Introduction

Bezem and Groote [5] incorporated several well-known and field-proven concepts such as precondition/effect notation and invariants in the framework of μCRL, aiming at a powerful verification methodology for distributed systems. Roughly, μCRL [9] can be considered as a dialect of ACP [2] extended with a formal treatment of data. The precondition/effect notation, as found in Unity [6] and I/O automata theory [15], is obtained in μCRL by restricting process expressions to a linear format; such expressions are called *linear process operators* (LPOs). Invariants are formulated in μCRL as predicates over linear operators.

In this paper, we formalise the μCRL versions of the notions LPO and invariant in Coq [8]. This is a proof checker, c.q. theorem prover, based on the Calculus of Constructions (higher-order type theory) [7], extended with inductive types [16]. The Coq representation of the LPOs and invariants has been placed on top of the basic proof system of μCRL that has already been implemented in Coq (see [17, 3]). Moreover, we have extended the theory of [5] with a *symbolic* representation format for LPOs. This format pays off in efficiency when dealing with LPOs in Coq.

We have chosen Coq as our proof development tool for the following reasons. First, we have some experience with the tool in which we already implemented the proof system of μCRL and carried out several computer-checked verifications (see [14, 11, 3]). Second, the system is highly expressive, i.e. supports higher-order reasoning. This is important because we need higher-order constructs for defining the notion of LPO. In fact, although Coq is rather expressive, we had to ask the tool builders to extend the functionality of Coq in order to deal with LPOs in a straightforward way. (In technical terms: we needed strong elimination on the type `Set` which was not possible in previous versions of Coq.) At last, the tool is well-supported and still under extension and improvement.

Our work contributes in the following aspects. First, we have built a uniform framework in which one can reason with invariants and process equations at the same time. As far as we know, this is new. Second, the notion of LPO turns out to be a vehicle for new verification techniques allowing for more automation. For instance in [10] the notions 'focus point' and 'convergent linear operators' are introduced, which simplify a considerable number of μCRL proofs. We have already formalised and integrated both these notions in the framework presented in this paper. We are currently preparing a new paper describing the new features just mentioned (see [13]).

Concluding, we see the work presented in this paper as a first step in the construction of a framework in which protocols like the Concurrent Alternating Bit Protocol [10] and the One Bit Sliding Window Protocol [4] can be verified (computer-checked) in a systematic and powerful way with as much computer-assistance as possible. Such situation would be a considerable improvement on the past as the existing correctness proofs of the two protocols mentioned above are at the moment handwritten, incomplete and error-prone.

The paper is organised as follows. In Section 2, we present the verification problem which is used as a running example throughout the paper. In Section 3, we present a subset of μCRL called pCRL (pico-CRL) and introduce the notions LPO and invariant. Then in Section 4, we formulate the verification problem in pCRL by using these two notions. In Section 5, we translate the theory of LPOs and invariants and the verification problem into Coq. In Section 6, we draw the conclusions of our work. In Appendix A, we give a brief explanation of the Coq system and give an example of a Coq proof session. In particular, a part of the Coq proof of the main theorem is displayed and explained. For people not familiar with Coq it is recommended to read the appendix before preceding with Section 5.

2 The verification problem

Let E be a set of values, totally ordered by a relation $\prec$. Then the input to the program given below is an element x from E and a finite array a with indices $0 \ldots n-1$ and elements taken from E.

$i := 0$; $k := n$;
while $i \neq k$ **do**
    $m := (i + k)$ div 2 ;
    if $a[m] \prec x$ **then** $i := m + 1$ **else** $k := m$ **fi**
od

div stands for division on natural numbers (i.e. $n \,\mathsf{div}\, m = \lfloor \frac{n}{m} \rfloor$). If the array a is sorted, the program can be used for detecting the element x in a. When the program terminates (it always does), then k can be interpreted as follows: if x occurs in a then $a[k] = x$. The same holds for variable i because after termination of the search procedure $i = k$.

As an illustration of our verification method, we will verify that the program satisfies the following invariant (property that holds in every program state):

1. $0 \leq i \leq k \leq n$,

2. $i > 0 \Rightarrow a(i{-}1) \prec x$,

3. $k < n \Rightarrow a(k) \succeq x$.

This invariant is rather abstract and also holds when a is not sorted.

This verification problem was proposed as a comparative case-study at the IPA workshop [12]. Every participant was asked to tackle the problem with her/his verification method.

3 LPOs and invariants

Here we recapitulate some terminology that has been introduced in [4].

We assume the existence of non-empty, disjoint sets with data elements, which are called *data types* and are denoted by letters D and E. Furthermore, we assume a set of many sorted operations on these sets, which are called *data operations*. There is one standard data type **Bool**, which consists of the elements $\top$ and $\bot$. We assume that the standard boolean operations are defined on **Bool**. We also assume the existence of a set $p\mathsf{Act}_{\delta\tau}$ that contains parameterised actions. Every element of $p\mathsf{Act}_{\delta\tau}$ is equipped with the data type of its parameter. The elements of $p\mathsf{Act}_{\delta\tau}$ are regarded as mappings from data types to processes. For technical convenience, δ (deadlock) and τ (silent step) are included in $p\mathsf{Act}_{\delta\tau}$. We assume that both δ and τ are equipped with the data type $\mathbb{1}$ containing only one element $\imath$ which is used as a dummy parameter. Furthermore, $\delta(\imath)$ and $\tau(\imath)$ are abbreviated by respectively δ and τ.

Definition 3.1. A $p\mathrm{CRL}^1$-*algebra* is a set $\mathbb{P}$, disjoint from the data types, with operations

$a : D \to \mathbb{P}$ (for all $a \in p\mathsf{Act}_{\delta\tau}$, D the data type of a)
$+, \cdot : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$
$\sum : (D \to \mathbb{P}) \to \mathbb{P}$ (for each data type D)
$\_ \lhd \_ \rhd \_ : \mathbb{P} \times \mathbf{Bool} \times \mathbb{P} \to \mathbb{P}$

satisfying the (conditional) equations A1–7, SUM1,3,4,5,11, Bool1, Cond1, Cond2 from Table 1. If the algebra also satisfies the equations B1 and B2 for branching bisimulation [18] it is called a $p\mathrm{CRL}^{1\tau}$-*algebra*.

By using the data type with one element and by using pairing, actions depending on zero or more than one data type can be simulated. Therefore, we use zero or more than one parameter whenever convenient.

Table 1 displays a subset of the axioms of μCRL [9] (micro-CRL), which we call pCRL (pico-CRL). The operations $+$ and $\cdot$ and equations A1–7 and B1,2 are standard for process algebra (see [2]) and therefore not explained. The operation $\_ \lhd \_ \rhd \_$ is the *then-if-else* operation. The sum operation $\sum$ over a data type D expresses that its argument $p : D \to \mathbb{P}$ may be executed with some data element d from D. Instead of $\sum(\lambda d{:}D.x)$ we generally write $\sum_{d:D} x$. Note that we use explicitly typed λ notation to denote mappings. If convenient we sometimes drop the explicit types. We use the convention that $\cdot$ binds stronger than $\sum$, followed by $\_ \lhd \_ \rhd \_$, and $+$ binds weakest.

The notion of a linear process operator as given below is an important concept underlying new developments in μCRL.

A1	$x+y=y+x$	SUM1	$\sum_{d:D} x = x$
A2	$x+(y+z)=(x+y)+z$	SUM3	$\sum p = \sum p + p(d)$
A3	$x+x=x$	SUM4	$\sum_{d:D}(p(d)+q(d)) = \sum p + \sum q$
A4	$(x+y)\cdot z = x\cdot z + y\cdot z$	SUM5	$\sum_{d:D}(p(d)\cdot x) = (\sum p)\cdot x$
A5	$(x\cdot y)\cdot z = x\cdot(y\cdot z)$	SUM11	$(\forall d \in D\ p(d)=q(d)) \Rightarrow \sum p = \sum q$
A6	$x+\delta = x$		
A7	$\delta\cdot x = \delta$	Bool1	$\neg(\top = \bot)$
B1	$c\cdot\tau = c$	Cond1	$x \lhd \top \rhd y = x$
B2	$c\cdot(\tau\cdot(x+y)+x) = c\cdot(x+y)$	Cond2	$x \lhd \bot \rhd y = y$

Table 1: $x, y, z \in \mathbb{P}$, c is τ or $a(d)$ for $a \in p\mathsf{Act}_{\delta\tau}$, $d \in D$ and $p, q : D \to \mathbb{P}$, $b \in \mathbf{Bool}$

Definition 3.2. A *linear process operator* (LPO) over data type D is an expression of the form

$$\Phi = \lambda p{:}D\to\mathbb{P}.\ \lambda d{:}D.\ \textstyle\sum_{i\in I}\sum_{e_i:E_i} c_i(f_i(d,e_i))\cdot p(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta$$

for some finite index set I, actions $c_i \in p\mathsf{Act}_\tau$, data types E_i, D_{c_i}, and functions $f_i : D\to E_i\to D_{c_i}$, $g_i : D\to E_i\to D$, $b_i : D\to E_i\to\mathbf{Bool}$.

We will give an example below. Note that, writing $I = \{1,\ldots,n\}$, we use a meta-sum notation $\sum_{i\in I} p_i$ for $p_1 + p_2 + \cdots + p_n$; the p_i's are called *summands*. The difference between the notation of the sum of μCRL and the meta-sum is the use of the colon versus the use of the membership symbol.

In [5] an LPO is defined as having also summands that allow termination. We have omitted these here, because they hardly occur in actual specifications and obscure the presentation of the theory. Moreover, it is not hard to add them if so required.

LPOs are defined having a single data parameter. The LPOs that we will consider generally have more than one parameter, but using cartesian products and projection functions, it is easily shown that this is an inessential extension. In the following, we will often write b for $b = \top$.

Definition 3.3. Let Φ be a LPO over data type D, and let $Inv : D\to\mathbf{Bool}$ be a boolean function. Inv is an *invariant* of Φ if for each summand

$$\sum_{e_i:E_i} c_i(f_i(d,e_i))\cdot p(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta$$

we have that $b_i(d,e_i) \wedge Inv(d) \Rightarrow Inv(g_i(d,e_i))$.

Below we present the notion of symbolic LPO which is a more efficient representation. In fact via this representation LPOs will be fed in Coq.

Definition 3.4. A *symbolic* LPO is a octuple $t = (D, I, E, D_c, c, f, g, b)$ where

- D is a domain (set),
- I is an index set,
- $E \equiv \lambda i.E_i$ is a mapping that assigns a domain E_i to each index $i \in I$,
- $D_c \equiv \lambda i.D_{ci}$ is a mapping that assigns a domain D_{ci} to each index $i \in I$,
- $c \equiv \lambda i.c_i$ is a mapping that assigns an action $c_i \in p\mathsf{Act}_{\delta\tau}$ to each index $i \in I$,
- $f \equiv \lambda i.f_i$ is a mapping (functional) that assigns a function $f_i : D \times E_i \to D_{ci}$ to each index $i \in I$,
- $g \equiv \lambda i.g_i$ is a mapping (functional) that assigns a function $g_i : D \times E_i \to D$ to each index $i \in I$,
- $b \equiv \lambda i.b_i$ is a mapping (functional) that assigns a function $b_i : D \times E_i \to \mathbf{Bool}$ to each index $i \in I$.

Definition 3.5. Let $t = (D, I, E, D_c, c, f, g, b)$ be a *symbolic* LPO, and

$$\begin{aligned} &\mathit{makeLPO}(t) := \\ &\qquad \lambda p{:}D \to \mathbb{P}. \\ &\qquad\qquad \lambda d{:}D.\, \textstyle\sum_{i \in I} \sum_{e_i : E_i} c_i(f_i(d, e_i)) \cdot p(g_i(d, e_i)) \lhd b_i(d, e_i) \rhd \delta. \end{aligned}$$

Then $\mathit{makeLPO}(t)$ is called the *concrete* LPO *representation* of t.

The next proposition expresses that every LPO can be represented in a symbolic way.

Proposition 3.6. *For every (concrete)* LPO Φ *there exists a* symbolic LPO $t = (D, I, E, D_c, c, f, g, b)$ *such that* $\Phi = \mathit{makeLPO}(t)$.

The notion of an invariant for symbolic LPOs is defined in the obvious way:

Definition 3.7. Let $t = (D, I, E, D_c, c, f, g, b)$ be an *symbolic* LPO and $\mathit{Inv} : D \to \mathbf{Bool}$ be a boolean function. Then Inv is an *invariant* of t if for each $i \in I$, $d : D$ and $e_i : E_i$ we have that $b_i(d, e_i) \wedge \mathit{Inv}(d) \Rightarrow \mathit{Inv}(g_i(d, e_i))$.

The next proposition expresses that it makes no difference whether you check an invariant on the LPO itself or on its symbolic representation.

Proposition 3.8. *Let* $t = (D, I, E, D_c, c, f, g, b)$ *be a* symbolic LPO. *then* $\mathit{Inv} : D \to \mathbf{Bool}$ *is an invariant of* t *if and only if* Inv *is an invariant of* $\mathit{makeLPO}(t)$.

4 Tackling the problem in μCRL

Below we have specified the search program in μCRL. The upper part defines the needed data types and the lower part the program itself.

$$
\begin{array}{lll}
\textbf{sort} & \textbf{Bool}, D, \mathit{Nat} & \\
\textbf{func} & \top, \bot \;:\!\rightarrow \textbf{Bool} & 0, \mathsf{n} \;:\!\rightarrow \mathit{Nat} \\
 & \neg \;: \textbf{Bool} \rightarrow \textbf{Bool} & \mathsf{S},\ \mathsf{div}_2 : \mathit{Nat} \rightarrow \mathit{Nat} \\
 & \wedge \;: \textbf{Bool} \times \textbf{Bool} \rightarrow \textbf{Bool} & \mathsf{a} \;: \mathit{Nat} \rightarrow D \\
 & < \;: \mathit{Nat} \times \mathit{Nat} \rightarrow \textbf{Bool} & \mathsf{x} \;:\!\rightarrow D \\
 & \prec \;: D \times D \rightarrow \textbf{Bool} & \\
\textbf{var} & m, n \;: \mathit{Nat} & \\
 & b \;: \textbf{Bool} & \\
\textbf{rew} & \neg\top = \bot & 0\,\mathsf{div}_2 = 0 \\
 & \neg\bot = \top & \mathsf{S}(0)\,\mathsf{div}_2 = 0 \\
 & b \wedge \top = b & \mathsf{S}(\mathsf{S}(n))\,\mathsf{div}_2 = \mathsf{S}(n\,\mathsf{div}_2) \\
 & b \wedge \bot = \bot & 0{<}\mathsf{S}(n) = \top \\
 & 0{+}n = n & n{<}0 = \bot \\
 & \mathsf{S}(m){+}n = \mathsf{S}(m{+}n) & \mathsf{S}(n){<}\mathsf{S}(m) = n{<}m \\
\textbf{act} & \mathit{ready} \;: \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \times \textbf{Bool} & \\
\textbf{proc} & X(i, m, k{:}\mathit{Nat};\ s{:}\textbf{Bool}) = & \\
 & \quad \tau \cdot X(i, (i{+}k)\,\mathsf{div}_2, k, \neg s) \quad \lhd\, i{<}k \wedge \neg s \quad \rhd\, \delta\ + & \\
 & \quad \tau \cdot X(m{+}1, m, k, \neg s) \quad \lhd\, i{<}k \wedge s \wedge \mathsf{a}(m){\prec}\mathsf{x} \rhd\, \delta\ + & \\
 & \quad \tau \cdot X(i, m, m, \neg s) \quad \lhd\, i{<}k \wedge s \wedge \mathsf{a}(m){\succeq}\mathsf{x} \rhd\, \delta\ + & \\
 & \quad \mathit{ready}(i, m, k, s) \cdot X(i, m, k, s) \quad \lhd\, i = k \quad \rhd\, \delta & \\
 & SEARCH(m{:}\mathit{Nat}) = X(0, m, \mathsf{n}, \bot) &
\end{array}
$$

We use the following abbreviations: $\neg b$ for $\neg(b)$, in-fix for $+$ and $<$, $x\,\mathsf{div}_2$ for $\mathsf{div}_2(x)$, $x{\geq}y$ for $\neg\,(x{<}y)$, $x{\succeq}y$ for $\neg\,(x{\prec}y)$, $b_1 \Rightarrow b_2$ for $\neg(b_1 \wedge \neg b_2)$, $x = y$ for $\neg(x{<}y) \wedge \neg(x{\geq}y)$.

For technical reasons we took some freedom in the μCRL translation of the search program (X), e.g. the original program given in Section 2 terminates while X does not. In X we have modelled termination implicitly by using a *ready* action. X is considered to be 'terminated' if it has performed its first *ready* action. After this it will be repeating the *ready* action forever while remaining in the same state (the *ready* action does not change the data parameters). We could also have treated termination explicitly by replacing the fourth summand of X by

$$\mathit{ready}(i, m, k, s) \lhd i = k \rhd \delta$$

However such termination summand does not fit in the LPO format as given in Definition 3.2. If we want, we can add termination summands to our framework, but this will cause a lot of technical overhead which does not really deepen our insights.

Below the invariant given in Section 2 is represented in μCRL by the function inv.

$$\textbf{func} \quad \mathsf{inv}_1 \;: \mathit{Nat} \times \mathit{Nat} \rightarrow \textbf{Bool}$$

$$
\begin{array}{ll}
 & \mathsf{inv}_2, \mathsf{inv}_3 \quad : \mathit{Nat} \to \mathbf{Bool} \\
 & \mathsf{inv}_4, \mathsf{inv} \quad : \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \times \mathbf{Bool} \to \mathbf{Bool} \\
\mathbf{rew} & \mathsf{inv}_1(i,k) = (0 \leq i \wedge i \leq k \wedge k \leq n) \\
 & \mathsf{inv}_2(i) = (i \geq 0 \to a(i-1) \prec x) \\
 & \mathsf{inv}_3(k) = (k < n \to a(k) \succeq x) \\
 & \mathsf{inv}_4(i,k,m,s) = (s \to m = (i{+}k)\,\mathsf{div}_2) \\
 & \mathsf{inv}(i,k,m,s) = (\mathsf{inv}_1(i,k) \wedge \mathsf{inv}_2(i) \wedge \mathsf{inv}_3(k) \wedge \mathsf{inv}_4(i,m,k,s))
\end{array}
$$

Note that $\mathsf{inv}_1(i,k) \wedge \mathsf{inv}_2(i) \wedge \mathsf{inv}_3(k)$ corresponds with the invariant given in Section 2. However, in order to get the proof done, we have loaded the invariant (inv) loaded with an extra conjunct (inv_4). Moreover, we shall need the following three data identities:

Lemma 4.1.

1. $i < k \to (i+k)\,\mathsf{div}_2 + 1 \leq k$,
2. $i < k \to i \leq (i+k)\,\mathsf{div}_2$,
3. $(i \leq k \leq n) \to (i+k)\,\mathsf{div}_2 \leq n$.

Theorem 4.2.

1. $\mathsf{inv}(0, m, \mathsf{n}, \bot)$, *i.e. the invariant holds in the initial state of* X,
2. inv *is an invariant of* X.

Proof.

1. Easy.
2. By Definition 3.3 we have to check the following four data formulas:

   (i) $i{<}k \wedge \neg s \wedge \mathsf{inv}(i,m,k,s) \to \mathsf{inv}(i, (i{+}k)\,\mathsf{div}_2, k, \neg s)$,
   (ii) $i{<}k \wedge s \wedge \mathsf{a}(m) {\prec} \mathsf{x} \wedge \mathsf{inv}(i,m,k,s) \to \mathsf{inv}(m{+}1, m, k, \neg s)$,
   (iii) $i{<}k \wedge s \wedge \mathsf{a}(m) {\succeq} \mathsf{x} \wedge \mathsf{inv}(i,m,k,s) \to \mathsf{inv}(i, m, m, \neg s)$
   (iv) $i = k \wedge \mathsf{inv}(i,m,k,s) \to \mathsf{inv}(i,m,k,s)$.

   (ii) and (iv) are easy. For proving (i), Lemma 4.1.1 is used. And (iii) is proved by Lemma 4.1.2 and 4.1.3.

❑

In Section 5, we formalise this theorem in Coq.

5 Formalisation in Coq

For a short introduction to Coq, see Appendix A. The complete code of the proof development can be obtained by contacting the authors.

5.1 μCRL

The action set $p\mathsf{Act}_{\delta\tau}$, which we will call `act` here, is represented as an inductive set in Coq:

```
Inductive act:Set :=  ready :act | tau:act | delta:act.
```

Below we introduce the process operators of pCRL in Coq. The parameters `alt, seq, cond, sum` correspond with the operators $+$, $\cdot$, $\_ \lhd \_ \rhd \_$, $\sum$, respectively.

```
Parameter proc :Set.
Parameter ia   :(A:Set)act->A->proc.
Parameter alt  :proc->proc->proc.
Parameter seq  :proc->proc->proc.
Parameter cond :proc->bool->proc->proc.
Parameter sum  :(A:Set)(A->proc)->proc.
```

Actually, `proc->proc->proc` is an abbreviation of `proc->(proc->proc)`, i.e. Coq associates the arrows to the right. We illustrate the application of the `ia` operator (interpretation action) by the following example: the parametrised action $ready(0)$ is represented in Coq by the term `(ia nat ready 0)`. Below, δ (deadlock) and τ (silent step) are defined by using a dummy argument (`one` corresponds with $\mathbb{1}$ and `dummy` with $\imath$).

```
Inductive  one:Set :=  dummy:one.
Definition Delta   := (ia one delta dummy).
Definition Tau     := (ia one tau   dummy).
```

Another example, the μCRL process $\tau \cdot \delta + \delta$ is represented by

```
(alt (seq Tau Delta) Delta)
```

In Coq, function application is usually denoted in 'prefix' style. In fact the term given above is an abbreviation of `((alt (seq Tau Delta)) Delta)`. In contrast with arrow types (`->`), the system associates function application to the left.

5.2 LPO

The index set is represented as a list as follows.

```
Inductive list [I:Set]  : Set :=
    nil:  (list I)
  | Cons: I->(list I)->(list I).
Syntactic Definition cons := (Cons ?).
```

The list is parametrised by an arbitrary index set `I`. As an illustration we give an example how this definition works: `(list act)` is the type of a list with elements from `act`, and the term `(Cons act ready nil)` represents the insertion of the `ready` action in the empty list. As Coq supports implicit typing, one may also write `(Cons ?  ready nil)`. The

Coq system knows that the place holder `?` represents `act`, which is the type of `ready`. Using a syntactic definition one can write the more compact `(cons ready nil)` for `(Cons ? ready nil)`.

Below the notion of a symbolic LPO (see Definition 3.4) is defined in Coq.

```
Record tupel [D,I:Set; E,Dc:I->Set] : Set :=
  make_tupel { Get_I  :  (list I);
               Get_c  :  I->act;
               Get_f  :  (i:I)D->(E i)->(Dc i);
               Get_g  :  (i:I)D->(E i)->D;
               Get_b  :  (i:I)D->(E i)->bool }.

Definition  Get_D    :=  [D,I:Set][E,Dc:I->Set][t:(tupel D I E Dc)]D.
Definition  Get_E    :=  [D,I:Set][E,Dc:I->Set][t:(tupel D I E Dc)]E.
Definition  Get_Dc   :=  [D,I:Set][E,Dc:I->Set][t:(tupel D I E Dc)]Dc.
```

This record construct is a new feature of Coq allowing the definition of records as is done in many programming languages. The record is a macro in the sense that the system translates it internally into an inductive definition which belongs to the kernel functionality of Coq. Note that in the fifth entry (`Get_b`) the type `bool` is used which is a standard data type of Coq. It has two constructors `true` and `false`.

In total, the record consists of nine fields: the four parameters `D, I, E, Dc` plus the five entries that are prefixed with `Get_`. The record given above has nine fields whereas the tuple given in Definition 3.4 has just eight fields. The reason of this is an extra field for a list representation of the index set. Often reasoning via lists is easier than reasoning via sets in Coq.

An example: assume that the identifiers `D, I, E, Dc, I_list, c, f, g, b` are already know to Coq and have the appropriate types. Then we can construct an LPO tuple as follows.

```
Definition  Phi := (make_tupel D I E Dc I_list c f g b).

Coq < Compute (Get_f D I E Dc Phi).
     = f
     : (i:I)D->(E i)->(Dc i)
```

By using syntactic definitions, e.g. we can suppress implicit type information and write more friendly terms as given below.

```
Syntactic Definition get_f := (Get_f ? ? ? ?).
Coq < Compute (get_f Phi).
     = f
     : (i:I)D->(E i)->(Dc i)
```

Below, the function *makeLPO* from Definition 3.5 is translated.

```
Definition MakeSummand :=
  [D,I:Set][E,Dc:I->Set][t:(tupel D I E Dc)]
    [i:I] [p:D->proc] [d:D]
      (sum (E i) [e_i:(E i)](cond (seq (ia (Dc i)
                                           (get_c t i)
                                           (get_f t i d e_i))
                                       (p (get_g t i d e_i)))
                                  (get_b t i d e_i)
                                  Delta)).
Syntactic Definition make_summand :=  (MakeSummand ? ? ? ?).

Definition MakeLPO :=
  [D,I:Set][E,Dc:I->Set][t:(tupel D I E Dc)]
    <(D->proc)->(D->proc)>Match (get_I t) with
            (* nil *)          [p:D->proc][d:D]Delta
            (* cons i l' *)    [i:I][l':(list I)]
                               [rec:(D->proc)->(D->proc)]
                                 [p:D->proc][d:D]
                                   (alt (make_summand t i p d) (rec p d))
                          end.
Syntactic Definition makeLPO :=  (MakeLPO ? ? ? ?).
```

The `MakeSummand` definition builts one single summand of the concrete LPO. The function `MakeLPO` iterates through the index list and constructs a (concrete) LPO by putting together the single summands that are returned by `MakeSummand` via alternative composition (`alt`). The iteration is done by `Match`-function which enables primitive recursion by induction on the constructors. In this case the constructors are `nil` and `cons`.

5.3 Invariant

The notion of invariant as given in Definition 3.7 can be represented in Coq in a straightforward way.

```
Definition invariant :=
 [D,I:Set][E,Dc:I->Set][t:(tupel D I E Dc)][Inv:D->bool]
 (i:I)(d:D)(e_i:(E i))
 (Inv d)=true -> (get_b t i d e_i)=true -> (Inv (get_g t i d e_i))=true.
Syntactic Definition Invariant := (invariant ? ? ? ?).
```

Note that the connective `/\` in Definition 3.7 has been replaced by an implication `->`. The latter representation is called the 'Curried' version (see Section A).

5.4 Data types

Below the Coq representation of the data types are given.

```
Require Export Bool.
```

By this line, a standard library of boolean functions is imported, e.g. `neg` (negation), `andb` (conjunction), and `implb` (implication on a boolean level). The $<$ (less than) on natural numbers is translated as

```
Recursive Definition lt_bool : nat->nat->bool :=
  0     (S n)     => true
| n     0         => false
| (S n) (S m)     => (lt_bool n m).
```

The `Recursive`-construct is a macro, i.e. internally it is translated into `Case` and `Fixpoint` constructs. An example of the latter two constructs is given by the following Coq definition of div_2.

```
Fixpoint div2 [n:nat] : nat :=
  <nat>Case n of 0 [p:nat]<nat>Case p of 0 [q:nat](S (div2 q)) end end.
```

The `Fixpoint` mechanism allows recursive calls in a function definition. The `Case` construct is used for distinguishing on the constructors. Unfortunately, the current version of Coq does not accept the more elegant:

```
(* Recursive Definition div2 : nat->nat :=
     0         => 0
   | (S 0)     => 0
   | (S (S n)) => n. *)
```

Coq ignores all text between the comment brackets `(* .... *)`.

The remaining data type definitions are straightforward and omitted.

5.5 The search program

The LPO $X(i, m, k : Nat;\ s : \mathbf{Bool})$ given in Section 4 is an abbreviation for

$$X(d : Nat \times Nat \times Nat \times \mathbf{Bool})$$

as LPOs are restricted to one parameter. The parameters are now packed with in a 4-tuple which can be used as a single parameter. The 4-tuple is defined as follows:

```
Record quadruple : Set
:= make       { get_i:  nat;
                get_m:  nat;
                get_k:  nat;
                get_s:  bool }.
```

Below the symbolic LPO representation of the search program is given in Coq code.

```
Section Specification.
Local  D             := quadruple.
Inductive I:Set      := i1:I  | i2:I  | i3:I | i4:I.
Local E :I->Set      := [i:I]<Set>Case i of one one one one end.
Local Dc:I->Set      := [i:I]<Set>Case i of one one one D  end.
Local c :I->act      := [i:I]<act>Case i of tau tau tau ready end.
Local I_list         := (cons i1 (cons i2 (cons i3 (cons i4 nil)))).

Local  f :(i:I)D->(E i)->(Dc i) :=
  [i:I][d:D]
     <[i:I](E i)->(Dc i)>Case i of
        [e:(E i1)]dummy
        [e:(E i2)]dummy
        [e:(E i3)]dummy
        [e:(E i4)](make (get_i d)  (get_m d) (get_k d) (get_s d))
end.

Local  g :(i:I)D->(E i)->D :=
  [i:I][d:D]
     <[i:I](E i)->D> Case i of
        [e:(E i1)](make (get_i d) (div2 (plus (get_i d) (get_k d)))
                        (get_k d) (neg (get_s d)))
        [e:(E i2)](make (S (get_m d))
                                  (get_m d) (get_k d) (neg (get_s d)))
        [e:(E i3)](make (get_i d) (get_m d) (get_m d) (neg (get_s d)))
        [e:(E i4)](make (get_i d) (get_m d) (get_k d) (get_s d))
end.

Local  b :(i:I)D->(E i)->bool :=
  [i:I][d:D]
     <[i:I](E i)->bool> Case i of
        [e:(E i1)](andb (lt_bool (get_i d) (get_k d)) (neg (get_s d)))
        [e:(E i2)](andb (andb (lt_bool (get_i d) (get_k d)) (get_s d))
                        (lt_Dom (a (get_m d)) x))
        [e:(E i3)](andb (andb (lt_bool (get_i d) (get_k d)) (get_s d))
                        (neg (lt_Dom (a (get_m d)) x)))
        [e:(E i4)](eq_nat (get_i d) (get_k d))
end.

Definition Phi  := (make_tupel D I E Dc  I_list c f g b).
Definition init := (make O m n false).
End Specification.
```

`Phi` is the symbolic LPO of the search program and `init` its initial state. The `Section` mechanism allows for local definitions. When `Local` is used in stead of `Definition` then the name of the definition is only visible within its surrounding section.

Here, the index list `I_list` has exactly the same elements as the index set `I`. However, one could also have taken a smaller list like `(cons i1 (cons i2 nil))`. In that case, the corresponding concrete LPO generated by `makeLPO` would only consist of two summands. Although safe, such approach is not recommended because then `i3, i4` have become dummy indici which obscure the specification of the symbolic LPO.

As an illustration, below the normal form of the term `(make_summand Phi i4)` is given. It represents the fourth summand of the concrete LPO representation of `Phi` (compare it with the μCRL representation given in Section 4).

```
[p:quadruple->proc]
 [d:quadruple]
  (sum one
    [_:one]
     (cond
       (seq
         (ia quadruple ready
           (make (get_i d) (get_m d) (get_k d) (get_s d)))
         (p (make (get_i d) (get_m d) (get_k d) (get_s d))))
       (eq_nat (get_i d) (get_k d)) Delta))
```

5.6 The inv function

Below the function inv : $D \Rightarrow$ **Bool** defined in Section 4 is represented in Coq.

```
Section Inv_func_sec.
Definition D := (get_D Phi).
Definition conj1 :=
  [d:D](andb (le_bool 0 (get_i d))
             (andb (le_bool (get_i d) (get_k d)) (le_bool (get_k d) n))).
Definition conj2 :=
  [d:D](implb (gt_bool (get_i d) 0) (lt_Dom (a (pred (get_i d))) x)).
Definition conj3 :=
  [d:D](implb (lt_bool (get_k d) n) (le_Dom x (a (get_k d)))).
Definition conj4 :=
  [d:D](implb (get_s d) (eq_nat (get_m d)
                                (div2 (plus (get_i d) (get_k d))))).
Definition Inv_func :=
  [d:D](andb (conj1 d) (andb (conj2 d) (andb (conj3 d) (conj4 d)))).
End Inv_func_sec.
```

In the definition of `conj2`, the predecessor function `pred` on natural numbers is used. This function is included in the standard library of Coq. Note that `implb` is an implication function on booleans.

Later, we will prove that `Inv_func` is an invariant of the search program `Phi`.

5.7 The verification problem

The main theorem of this paper is represented by the following two Coq expressions:

```
Goal (Inv_func init)=true.

Goal (Invariant Phi Inv_func).
```

The first goal is easy to prove in Coq and will not be discussed here. The second goal is proved as follows. After typing, the two tacticals

```
Unfold invariant ; Simpl.
Intro index ; Elim index ; Induction d
            ; Unfold Inv_func conj1 conj2 conj3 conj4
            ; Simpl ; Intros i m k s.
```

the proof engine returns four subgoals which correspond with the four data formulas given in the proof outline of Theorem 4.2.2. The semi-colon is used for composing tactics, i.e. the tactical `tac1 ; tac2` applies the tactic `tac2` to all the subgoals generated by `tac1`.

The Coq representation of the first subgoal is as follows:

```
(andb (andb (le_bool i k) (le_bool k n))
        (andb (implb (gt_bool i 0) (lt_Dom (a (pred i)) x))
          (andb (implb (lt_bool k n) (le_Dom x (a k)))
            (implb s (eq_nat m (div2 (plus i k)))))))
       =true
       ->(andb (lt_bool i k) (neg s))=true
          ->(andb (andb (le_bool i k) (le_bool k n))
              (andb (implb (gt_bool i 0) (lt_Dom (a (pred i)) x))
                (andb (implb (lt_bool k n) (le_Dom x (a k)))
                  (implb (neg s)
                    (eq_nat (div2 (plus i k)) (div2 (plus i k)))))))
             =true
```

The corresponding proof session of this subgoal is similar to the one given in Section A.3. Note that in the subgoal above `implb` denotes implication on booleans (`bool`) and `->` denotes implication on propositions (`Prop`).

6 Discussion

The Coq proof consists of about 200 commands, and it takes 3 minutes to run on a SUN SPARCstation. The complete proof development (including the formalisation of the proof theory) comprises 14,6 Kb. These figures only give a raw indication as we neither did optimise the length or the execution time of the proof. We only used tacticals (operators

that combine tactics for writing synthetic proof scripts) when there was structural repetition in the proof commands. However, we could have written many more proof scripts for minimising the proof length. We experienced that the use of tacticals reduces the proof code but mostly increases execution time.

In several places, the readability of the Coq code given in this paper is not optimal yet. For instance, the definition of `div2` given in Section 5 is rather cryptic. It would be an improvement when, in future releases of the Coq system, `div2` could be reformulated by a `Recursive Definition` similar to `lt_bool`. Furthermore, readability would be improved in case in-fix or mix-fix notation was used, e.g. writing `Delta * Delta + Tau` instead of `(alt (seq Delta Delta) Tau)`. In fact, the new release of Coq provides facilities supporting such notation. In particular, the user may define her/his own rules for extending the grammar of the Coq language. However, due to a lack of time and the fact that writing appropriate grammar rules is not completely trivial, we have postponed the matter of pretty syntax as future work.

References

[1] D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors. *Proceedings of the International Workshop on Semantics of Specification Languages,* Utrecht, The Netherlands. Workshops in Computing, Springer-Verlag, 1993.

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

[3] M.A. Bezem, R. Bol, and J.F. Groote. Formalizing process algebraic verifications in the calculus of constructions. Computing Science Report 95-02, Eindhoven University of Technology, 1995.

[4] M.A. Bezem and J.F. Groote. A correctness proof of a one-bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.

[5] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proceedings of the* 5^{th} *Conference on Theories of Concurrency, CONCUR '94,* Uppsala, Sweden, August 1994, volume 836 of *Lecture Notes in Computer Science*, pages 401–416. Springer-Verlag, 1994.

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

[7] T. Coquand and G. Huet. The calculus of constructions. *Information and Control*, 76:95–120, 1988.

[8] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. The Coq proof assistant reference manual. Version 5.10. Technical report, INRIA - Rocquencourt — CNRS - ENS Lyon, July 1995.

[9] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In Andrews et al. [1], pages 231–250.

[10] J.F. Groote and J. Springintveld. Focus points and convergent process operators: A proof strategy for protocol verification. To appear as Technical Report, Logic Group Preprint Series, Utrecht University, 1995.

[11] J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. A case study in computer checked verification. Technical Report 100, Logic Group Preprint Series, Utrecht University, October 1993.

[12] `http://www.win.tue.nl/win/cs/ipa`.

[13] H. Korver and M.P.A. Sellink. Formalising convergent process operators and focus points in Coq. In preparation.

[14] H. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. In M. Hagiya and J.C. Mitchel, editors, *Proceedings of the* 2^{nd} *International Symposium on Theoretical Aspects of Computer Software, TACS '94,* Sendai, Japan, volume 789 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 1994.

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

[16] C. Paulin-Mohring. Inductive definitions in the system Coq. Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the* 1^{st} *International Conference on Typed Lambda Calculi and Applications, TLCA '93,* Utrecht, The Netherlands, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.

[17] M.P.A. Sellink. Verifying process algebra proofs in type theory. In Andrews et al. [1], pages 315–339.

[18] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618, 1989.

A The Coq system

In this appendix we briefly explain the ideas behind the use of type theory for proof checking purposes and in particular use of the interactive proof construction program Coq. We use `typewriter style` for terms and types. Furthermore we use the conventions of the ASCII Coq-interface.

A.1 The type theoretic approach

A *type system* is a set of rules to assign types to the set of untyped λ-terms. Variables have a type by declaration (stored in a context). A λ-term `f` that can be applied to a λ-term `a` of type `A` yielding a λ-term `f a` of type `B` is of type `A->B`. In general, the type of application `f a` may depend on `a`. If `f a` is of type `B[x:=a]` (the result of substituting `a` for each occurrence of `x` in `B`) then `f` is of type `(x:A)B`. For readability reasons we write `A->B` rather than `(x:A)B` if `x` does not occur as a free variable in `B`.

One can think of types as *sets* and of the colon as membership. Thus, if `f:A->B` then `A` and `B` represent sets A and B and `f` represents a function from A to B. A function $g : A \times B \longrightarrow C$, which is actually a subset of $(A \times B) \times C$ can then be represented by identifying g with its "Curried" version $Curry(g) \subseteq A \times (B \times C)$, which yields a term of type `A->(B->C)`.

Types can also be seen as *propositions*. The colon is then red as "is a proof of". This means that we identify propositions with the set of its proofs. A term `f:A->B` is then an objects that takes a proof of proposition A and returns a proof of proposition B. This is precisely the intuitionistic view on a proof of the logical implication $A \Rightarrow B$. A function $g : A \times B \longrightarrow C$ takes a pair $\langle a, b\rangle \in A \times B$ (i.e. a proof of $A \wedge B$) and returns a proof of C and is hence a proof of $(A \wedge B) \Rightarrow C$.

In order to distinguish between those two interpretations we use the predefined types `Set` and `Prop` of Coq for types that represent sets and propositions respectively. For instance if we introduce

```
Parameter nat : Set.                Parameter O : nat.
Parameter Q   : nat->Prop.          Parameter S : nat->nat.
```

then `nat` represents the set $\mathbb{N}$ of natural numbers and `O`, `(S O)`, `(S (S O))`,... represent the elements of $\mathbb{N}$. The term `Q` represents a function from natural numbers to propositions, i.e. a predicate over $\mathbb{N}$.

In Coq one can introduce abbreviations by using the `Definition`-mechanism. Definitions are used to improve the readability by choosing intuitive names for terms. E.g.

```
Definition ONE   := (S O).          Definition id_nat := [x:nat]x.
Definition TWO   := (S ONE).        Definition add2   := [x:nat](S (S x)).
Definition THREE := (S TWO).
```

Formalising the problem in Coq is now a matter of introducing such parameters and definitions. Thus obtaining inhabitants of `Set` for the data types and inhabitants of `Prop` for the logical expressions. Definition in Coq can also be *inductive*. Since we extensively used this facility we briefly explain the idea behind inductive definitions.

A.2 Induction and recursion

Given a term `a` of type `(Q O)` and a term `g` of type `(x:nat)(Q x)->(Q (S x))` we can construct proofs for `Q x` for arbitrary closed term `x` of type `nat`, namely:

```
a                       : Q O
g O a                   : Q ONE
g ONE (g O a)           : Q TWO
g TWO (g ONE (g O a)) : Q THREE
```

and so on. However, we can not prove `(x:nat)(Q x)`. In the presentation above it is not formalised that `nat` does not contain any other terms than those constructed from `O` and `S`. If we want to formalise this we have to use the following alternative:

```
Inductive Definition nat : Set = O:nat | S:nat->nat.
```

The latter alternative allows the definition of recursive functions by means of a `Case`-construction. If we define

```
Definition F := [n:nat](<A>Case n of a g end).
```
(1)

where `a` of type `A` and `g` of type `nat->A` then `F` has the following reduction behaviour:

```
F O      ↠ (<A>Case  O    of a g end)  ↠ a
F (S x)  ↠ (<A>Case (S x) of a g end)  ↠ g x
```

For instance the predecessor on natural numbers (satisfying $pred(0) = 0$) is defined by:

```
Definition pred := [n:nat](<nat>Case n of O id_nat end).
```
(2)

The function `g` in (1) may depend on `F` itself. Namely, it is allowed to use `F` on "lower" values. In the example (2) above this dependency does not occur (`id_nat` does not make use of `pred`). In the following example it does:

```
Definition mul2 := [n:nat](<nat>Case n of O [x:nat](add2 (mul2 x)) end).
```

The definition above does not uniquely define a function. Actually it is not even a definition but just a recursive equation. The intended function (multiplication by 2) is the least fixpoint of the functional

```
[F:nat->nat] F := [n:nat](<nat>Case n of O [x:nat](add2 (F x)) end).
```

i.e. the least function that satisfies the recursive equation. In Coq this has to be made explicit by typing:

```
Fixpoint mul2 [n:nat]:nat := <nat>Case n of O [x:nat](add2 (mul2 x)) end.
```

A.3 Example of a Coq proof session

In this subsection we give the reader a flavour of a Coq proof session. We discuss the construction of a proof term by the type

```
(i,k:nat)(lt_bool i k)=true->(le_bool (S (div2 (plus i k))) k)=true.
```
(3)

which is the Coq representation of Lemma 4.1.1. Here `a=b` is pretty printing for `eq A a b`, where `eq` stands for polymorphic equality and `A` is the type of both `a` and `b`. To prove (3) with ordinary nested induction (to `i` and `k`) is problematic because the term `div2` does not reduce in case of *one* successor in its argument. The simplest solution is to use double induction. In that case a successor is substituted for both `i` and `k` at the same time and hence a double successor occurs in the argument of `div2`. Thus, we first prove the double induction principle:

```
(R:nat->nat->Prop)
  ((x:nat)(R O x))->
    ((x:nat)(R (S x) O))->
      ((x,y:nat)(R x y)->(R (S x) (S y)))->
        (x,y:nat)(R x y).
```

This can straightforwardly be proved by using nested induction. We conclude the proof session with the command `Save double_ind`, which introduces `double_ind` as an abbreviation for the constructed proof term. Now we can use double induction by typing:

```
Intros i k ; Pattern i k ; Apply double_ind.
```

which means that we first assume `i` and `k`, then transform[1] the goal to an expression of the form `(R i k)` with `R` of type `nat->nat->Prop`, and then apply `double_ind` with `R` instantiated with the relation occurring in the goal. This results into three subgoal, corresponding to `(x:nat)(R O x)`, `(x:nat)(R (S x) O)` and `(x,y:nat)(R x y)->(R (S x) (S y))`. The first of these three goals

```
(x:nat)(lt_bool O x)=true->(le_bool (S (div2 (plus O x))) x)=true.        (4)
```

is proved with yet another induction principle, namely

```
Goal (P:nat->Prop)
       (P O)->
         (P ONE)->
           ((x:nat)(P x)->(P (add2 x)))->
             (x:nat)(P x).
```

This latter induction principle is proved with induction loading and the ordinary induction principle on natural numbers. We saved the latter induction principle under the name `two_ind`. Applying the command `Intro x ; Pattern x ; Apply two_ind` on subgoal (4) results then in three subgoals. The first two reduce to `true=true` and are solved automatically. The last subgoal

```
(x:nat)
  ((lt_bool O x)=true ->
  (le_bool (S (div2 (plus O x))) x)=true)
   ->
  ((lt_bool O (add2 x))=true ->
  (le_bool (S (div2 (plus O (add2 x)))) (add2 x))=true)
```

[1] `Pattern a` transform a goal `M[x:=a]` to the redex `([x:A]M)a`

reduces by unfolding the definitions for `plus`, `div2` and `add2` to

```
(x:nat)
  ((lt_bool O x)=true ->
  (le_bool (S (div2 x)) x)=true)
   ->
  ((lt_bool O (add2 x))=true ->
  (le_bool (S (div2 x)) (S x))=true)
```

which follows immediately from the following two general facts

```
(x,y:nat)(le_bool x y)=true -> (le_bool x (S y))=true.

(x:nat)(le_bool (div2 x) x)=true.
```

which were added and proved without problems.

Thus, we solved the first of the three subgoal generated by applying double induction on (3). The second subgoal was proved by `Auto`, and the third subgoal followed immediately from the induction hypothesis after some `Rewrite`-steps needed to go from `(plus x (S y))` to `(S (plus x y)`, which is not just a reduction step because `plus` is defined by recursion on the first argument.