Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

Algebraic Transformations
Verification of Basic LOTOS processes

Section 2.4 of "Correctness Preserving Transformation"
ESPRIT Project 2304 LOTOSPHERE

Rocco De Nicola, Paola Inverardi, Monica Nesi

Nota Interna B4-34
Agosto 1990

Section 2.4

Verification of Basic LOTOS Processes

This section presents the approach to a verification environment which supports correctness preserving transformations on terms of the Basic LOTOS process algebra based on observational equivalence.

2.4.1 Motivations

The need for formal specifications of concurrent communicating systems and of their properties is generally acknowledged. Together with it, automatic or semi-automatic tools for supporting the analysis of concurrent systems are strongly advocated. This is, both to make analysis possible and to ensure correctness of specifications.
LOTOS can be used to describe concurrent systems at different levels of abstraction. It is equipped with a notion of equivalence, observational equivalence, and a notion of preorder, testing preorder, which can be used to prove that two different LOTOS specifications are equivalent when ignoring "uninteresting" details and to prove that a low level specification is a satisfactory implementation of a more abstract one and thus to perform part of the analysis.
The formal semantics of LOTOS is defined in terms of labelled transition systems which are factorized by observational equivalence or testing preorders. These relations give rise to a series of interesting (in)equational laws which can be used to refine specifications or to transform them without changing their semantics.
Our approach to the development of a verification system relies on the idea of taking advantage of the axiomatic presentation of behavioural equivalences. It can be exploited in two different ways. On one hand, the axiomatic presentation can be seen as an equational theory and its associated canonical term rewriting system (which, if existing, can be obtained by means of a completion algorithm [K87]) can be used to prove that two processes are equivalent by checking whether they can be reduced to "similar" normal forms. On the other hand, the axiomatic presentation can be used to perform elementary transformations, which are guaranteed to be correct, by applying single axioms on demand.
Given the axiomatic presentation for behavioural equivalences, the implementation of our verification system relies on *rewriting techniques*. An *equational approach* to "execute" the complete axiomatizations has been adopted and techniques borrowed from logic-functional programming are used: both the operational semantics and the axioms for behavioural equivalences are seen as Horn theories. This makes it possible to manage at metalevel (via metaprogramming) the intertwining between operational semantics and behavioural equivalences according to different user defined proof strategies. In this way we obtain those functionalities that will be the basis for developing a language to define strategies.
Indeed, we think that users should have a high control over the verification process, i.e. a verification system should allow users to perform their proof by using some automatic tools and controlling their interactions in a flexible way. Thus, we aim at taking advantage of all three views of specifications, namely the transitions, the operationally defined equivalences

and, whenever possible, the axioms which completely characterize the equivalences. This would allow users to define their own verification strategies and move from one view to another, to use each time the most convenient one. For example, we want to offer the possibility of executing the operational semantics and reducing terms by means of behavioural equivalences within a flexible and open-ended system, that makes tools and not policies available to perform verification.
In general, it is not fruitful to rely on a completely automatic approach to verification, even in the simple case of finite processes. A system which leaves some crucial decisions to the user is preferable. The validity of a *mixed* approach to the analysis and verification of concurrent systems has been confirmed by the experience we gained with CCS. In fact, before dealing with LOTOS, we have experimented our ideas on a verification system for CCS specifications which has been enriched with the complete axiomatizations of various behavioural equivalences [DIN89]. Several problems have been encountered when trying to execute the axiomatizations via the associated term rewriting system. In fact, the axiomatizations of well-known behavioural equivalences, like observational and testing congruences [HM85, DH84], do not lead to finite canonical rewriting systems, i.e. the completion process diverges while generating an infinite set of rules. This is the main limit of a straightforward application of the term rewriting system theory to the execution of behavioural axiomatizations.
As far as observational congruence of finite CCS is concerned, we have recovered from the divergence of the completion process within the framework of term rewriting system theory itself. We have defined a rewriting strategy [IN89] which is able to verify the observational congruence of two finite CCS specifications without performing any completion.

Proving correctness of LOTOS processes

For a significant sub-calculus of LOTOS, we can single out two sets of laws for the two behavioural relations included in the final text of the International Standard [B88], namely observational equivalence and testing preorder. We can use these sets of laws as the basis for an equational reasoning on LOTOS specifications of concurrent systems.
We consider a subset of the axioms which have been defined for the observational congruence. The subset under consideration represents a set of axioms which is able to rewrite a finite basic LOTOS behaviour into an observational congruent one containing only occurrences of the primitive LOTOS operators, namely ";" , "[]" and "**stop**" and a set of axioms for the primitive operators. This set of axioms is correct and fully characterizes observational congruence for finite basic LOTOS; thus we can use CCS signature to interpret basic LOTOS and apply all tools and strategies, at present available in the CCS verification system, to basic LOTOS behaviours as well.

2.4.2 The LOTOS verification environment

The LOTOS verification system at present includes the following Prolog modules:
- a *Syntax Analyzer* which allows the user to input specifications according to the LOTOS syntax;
- a module *Operational Semantics* that implements the inference rules of LOTOS interleaving semantics and permits determining the sequences of actions a LOTOS behaviour may perform and the behaviour into which the original one is rewritten after each sequence;
- a module *Observational Axioms* which can be used to apply, on demand, the axioms for observational equivalence of finite basic LOTOS as single rewrite rules;
- a module *Observational Equivalence* which implements a term rewriting system associated to the axiomatic presentation of observational equivalence and permits several rewriting steps to be applied automatically on a LOTOS behaviour.

The actual version of the system has been extended with two strategies for verifying equivalences of recursive (finite state) LOTOS behaviours. It is worth noting that both

strategies permit dealing with recursive behaviours without using any rule for recursion, either in the operational semantics or in the observational axiomatization. These two strategies essentially apply the same rewriting process on a LOTOS behaviour and the following three rewriting steps can be performed by relying on the modules described above:

- single steps of the operational semantics are applied by using the module *Operational Semantics*;
- intermediate behaviours are reduced by using the module *Observational Equivalence*;
- searches are performed for sub-behaviours which can be replaced by process identifiers whenever these are associated to equivalent behaviours; this permits folding the current behaviour and dealing with infinite terms.

These three steps are repeatedly performed till a stop condition is met. The used termination condition of the rewritings above gives rise to the two strategies, which serve different purposes.

The first strategy, *LOTOS Verification*, can be used to verify that a behaviour B is a correct implementation of a specification S. It applies the above steps till B has been reduced to a form provably equivalent to S or till it proves that all possible transformations have been tried and that the two behaviours are different.

The second strategy, *LOTOS Reduce*, can be used to explore the behaviour of a given specification B and to transform it into a "minimal" one. The rewriting steps are applied until only previously computed transitions can be executed.

Another, obvious, strategy can be obtained by combining the two strategies above.

Actual experimentation on the implementation of the LOTOS verification system in Quintus Prolog on a Sun-3 Workstation is under way and the architecture of the actual version is shown in Figure 2.4.2.1.

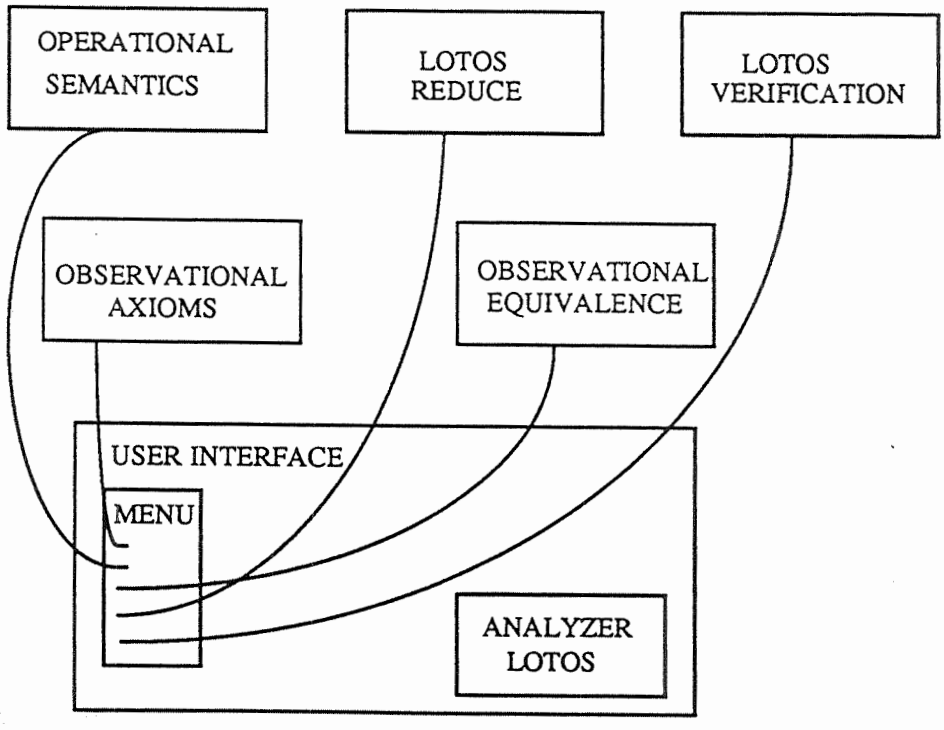

Figure 2.4.2.1 - The architecture of the LOTOS verification environment

Basic axioms for LOTOS observational equivalence

In the following we briefly recall the definition of observational equivalence and we list the set of axioms our system is based on.
Observational equivalence is based on the notion of bisimulation [HM85]. Loosely speaking, two behaviours B_1 and B_2 are considered as equivalent, written $B_1 \approx B_2$, if and only if there exists a relation $\Re$, called *bisimulation*, which contains the pair $\langle B_1, B_2 \rangle$ and guarantees that B_1 and B_2 are able to perform equal sequences of visible actions evolving to equal (up to $\Re$) behaviours.
Hereto, we will use the following conventions to talk about sequences of actions and sequences of visible actions:

- $B_1 \Rightarrow B_2$, ε being the null string of $(G \cup \delta)^*$, stands for $B_1 \xrightarrow{i}{}^{n} B_2$, $n \geq 0$;
- $B_1 \Rightarrow^{g^+} B_2$, stands for there exist B' and B" such that $B_1 \Rightarrow B' \xrightarrow{g^+} B'' \Rightarrow B_2$;
- $B_1 \Rightarrow^{s} B_2$, $s = g^+_1 ... g^+_n \in (G \cup \delta)^*$, stands for there exist B'_i, $0 < i < n$, such that $B_1 = B'_0 \Rightarrow^{g^+_1} B'_1 \Rightarrow^{g^+_2} \Rightarrow ... \Rightarrow^{g^+_n} \Rightarrow B'_n = B_2$;

Definition (*bisimulation and observational equivalence*)
1. If $\Re$ is a relation over LOTOS behaviours, then Ψ, a function from relations to relations, is defined as follows: $\langle B_1, B_2 \rangle \in \Psi(\Re)$ if, for all $s \in (G \cup \delta)^*$,
   i) whenever $B_1 \Rightarrow^{s} B'_1$ then there exists B'_2, $B_2 \Rightarrow^{s} B'_2$ and $\langle B'_1, B'_2 \rangle \in \Re$
   ii) whenever $B_2 \Rightarrow^{s} B'_2$ then there exists B'_1, $B_1 \Rightarrow^{s} B'_1$ and $\langle B'_1, B'_2 \rangle \in \Re$.
2. A relation $\Re$ is a *bisimulation* if $\Re \subseteq \Psi(\Re)$.
3. Relation $\approx$, defined as $\approx = \cup \Re \mid \Re \subseteq \Psi(\Re)$, is called *observational equivalence*. ♦

A correct, but not complete, set of axioms of observational congruence for LOTOS has been defined [B88]. In the following we give a set of axioms that defines a rewriting process of finite basic LOTOS behaviours into behaviours containing only occurrences of the primitive LOTOS operators, namely ";" , "[]" and "**stop**". Such rewriting process is correct and complete with respect to observational congruence.

Exit
T1. **exit** = δ; **stop**

Choice
C1. B1 [] B2 = B2 [] B1
C2. B1 [] (B2 [] B3) = (B1 [] B2) [] B3
C3. B [] **stop** = B
C4. B [] B = B

Relabelling
R1. **stop**[S] = **stop**
R2. (B1 [] B2)[S] = B1[S] [] B2[S]
R3. (μ^+; B)[S] = S(μ^+); B[S]

Hiding ($G \supseteq A$)
H1. **hide** A **in stop** = **stop**
H2. **hide** A **in** B1 [] B2 = (**hide** A **in** B1) [] (**hide** A **in** B2)
H3. **hide** A **in** μ^+; B =
$$\begin{cases} \mu^+; (\textbf{hide A in B}) & \text{if } \mu^+ \notin A \\ \textbf{i}; (\textbf{hide A in B}) & \text{if } \mu^+ \in A \end{cases}$$

Enabling
E1. **stop** >> B = **stop**
E2. (B1 [] B2) >> C = (B1 >> C) [] (B2 >> C)

E3. $(\mu^+; B) >> C = \begin{cases} \mu^+; (B >> C) & \text{if } \mu^+ \neq \delta \\ i; C & \text{if } \mu^+ = \delta \end{cases}$

Disabling

D1. **stop** [> B = B

D2. (B1 [] B2) [> C = (B1 [> C) [] (B2 [> C)

D3. $(\mu^+; B) [> C = \begin{cases} C [] \mu^+; (B [> C) & \text{if } \mu^+ \neq \delta \\ (\mu^+; B) [] C & \text{if } \mu^+ = \delta \end{cases}$

Parallel Composition

Notation: $B_1 [] B_2 [] ... [] B_n = [] B_1, ..., B_n = [] S$ where $S = B_1, ..., B_n$. The behaviour **stop** is denoted by means of empty summation [] .

P1. If $B = [] \mu_i^+; B_i \mid i \in I$, $C = [] \nu_j^+; C_j \mid j \in J$ and $G \supseteq A$,

$B \,|[A]|\, C =$ $[] \mu_i^+; (B_i \,|[A]|\, C) \mid \mu_i^+ \notin A \cup \delta,\ i \in I$

$[]$ $[] \nu_j^+; (B \,|[A]|\, C_j) \mid \nu_j^+ \notin A \cup \delta,\ j \in J$

$[]$ $[] \mu_i^+; (B_i \,|[A]|\, C_j) \mid \mu_i^+ = \nu_j^+,\ \mu_i^+ \in A \cup \delta,\ i \in I, j \in J$

Interleaving (A = Ø)

I1. $B \,|||\, C = B \,|[\,]|\, C$

Full Syncronization (A = G)

S1. $B \,||\, C = B \,|[G]|\, C$

Observational Congruence Laws

A1. $\mu; i; B = \mu; B$

A2. $B [] i; B = i; B$

A3. $\mu; (B1 [] i; B2) [] \mu; B2 = \mu; (B1 [] i; B2)$

2.4.3 Verification examples

Let us now show an application of the strategies mentioned above by means of two simple examples. We will apply the first strategy to a LOTOS description of a simple reader-writer problem, while the second strategy will be used to explore the behaviour of a LOTOS description of a vending machine. While we report the whole terminal session for the first example, we will only sketch the session for the second example.

The reader-writer problem

Given two processes Reader and Writer, we have to guarantee their mutual exclusive access to a common resource. We use the gates *xb*, *xe* to denote (respectively) the beginning and the end of the task of a process *X*. A mutual exclusive and nondeterministic access of processes *X*, *Y* to a common resource can be defined by the following basic LOTOS specification:

```
process Spec[xb, xe, yb, ye] :=
        (i; xb; xe; Spec[xb, xe, yb, ye]) [] (i; yb; ye; Spec[xb, xe, yb, ye])
endproc
```

Let us now consider a possible implementation of the above specification. We introduce a semaphore S to control access to the common resource and use the gates *px*, *vx* to denote (respectively) a 'p' and a 'v' operation on S of the process *X*:

```
process Impl[rb, re, wb, we] :=
        hide pr, vr, pw, vw in
        (S[pr, vr, pw, vw] |[pr, vr, pw, vw]|
                                (Reader[pr, vr, rb, re] ||| Writer[pw, vw, wb, we]))
where
        process S[px, vx, py, vy] :=
                (px; vx; S[px, vx, py, vy]) [] (py; vy; S[px, vx, py, vy])
        endproc
        process Reader[px, vx, xb, xe] :=
                px; xb; xe; vx; Reader[px, vx, xb, xe]
        endproc
        process Writer[py, vy, yb, ye] :=
                py; yb; ye; vy; Writer[py, vy, yb, ye]
        endproc
endproc
```

Following the first verification strategy, we can prove that Impl is a correct implementation of Spec with respect to observational equivalence (≈), i.e. we can prove:

Impl[rb, re, wb, we] ≈ (i; rb; re; Impl[rb, re, wb, we]) [] (i; wb; we; Impl[rb, re, wb, we])

Below, we report a sample of the interaction between the user (bold typed) and the system (plain text), sometimes interrupted by few comments (written in italic), which leads to the proof of our statement.

```
?- LOTOS verification.
/* The environment with the definitions of Impl, S, R, W and Spec is typed in */
Enter a process identifier: Impl
Enter a process: hide pr,vr,pw,vw in (S|[pr,vr,pw,vw]|(R|||W))
Enter a process identifier: S
Enter a process: pr;vr;S[]pw;vw;S
Enter a process identifier: R
Enter a process: pr;rb;re;vr;R
Enter a process identifier: W
Enter a process: pw;wb;we;vw;W
Enter a process identifier: Spec
Enter a process: i;rb;re;Impl[]i;wb;we;Impl
Enter a process identifier:
/*  indicates that all the needed terms have been entered */
/* we are prompted for two processes to be tested for equivalence */
Enter the first process: Impl
Enter the second process: Spec

Test of observational congruence: failed

Trying to transform Impl into Spec ...

Impl  =  hide pr,vr,pw,vw in (S|[pr,vr,pw,vw]|(R|||W))

Executing the operational semantics ...

Impl  =  (  i;hide pr,vr,pw,vw in (vr;S|[pr,vr,pw,vw]|(rb;re;vr;R|||W))
         [] i;hide pr,vr,pw,vw in (vw;S|[pr,vr,pw,vw]|(R|||wb;we;vw;W)))

There are no reductions on Impl

Testing whether some subexpressions can be replaced by process identifiers
```

```
 ....
no subexpression has been found.

Test of observational congruence: failed

Executing the operational semantics ...

Impl  =  (  i;rb;hide pr,vr,pw,vw in (vr;S|[pr,vr,pw,vw]|(re;vr;R|||W))
         [] i;wb;hide pr,vr,pw,vw in (vw;S|[pr,vr,pw,vw]|(R|||we;vw;W)))

There are no reductions on Impl

Testing whether some subexpressions can be replaced by process identifiers
 ....
no subexpression has been found.

Test of observational congruence: failed

Executing the operational semantics ...

Impl  =  (  i;rb;re;hide pr,vr,pw,vw in (vr;S|[pr,vr,pw,vw]|(vr;R|||W))
         [] i;wb;we;hide pr,vr,pw,vw in (vw;S|[pr,vr,pw,vw]|(R|||vw;W)))

There are no reductions on Impl

Testing whether some subexpressions can be replaced by process identifiers
 ....
no subexpression has been found.

Test of observational congruence: failed

Executing the operational semantics ...

Impl  =  (  i;rb;re;i;hide pr,vr,pw,vw in (S|[pr,vr,pw,vw]|(R|||W))
         [] i;wb;we;i;hide pr,vr,pw,vw in (S|[pr,vr,pw,vw]|(R|||W)))

There are reductions on Impl

Impl  =  (  i;rb;re;hide pr,vr,pw,vw in (S|[pr,vr,pw,vw]|(R|||W))
         [] i;wb;we;hide pr,vr,pw,vw in (S|[pr,vr,pw,vw]|(R|||W)))

Testing whether some subexpressions can be replaced by process identifiers
 ....
Replaceable subexpressions have been found:

Impl = hide pr,vr,pw,vw in (S|[pr,vr,pw,vw]|(R|||W))

Folding
 ....
Impl  =  (i;rb;re;Impl[]i;wb;we;Impl)

Test of observational congruence: ok!

The processes you entered are observational congruent.
```

A candy machine

Let us now assume that we are provided with two machines: a Slot Machine and a Fair Machine. They are defined in the following way. The Slot Machine takes as input a dime and decides internally whether to return either the same dime or a quarter or a piece of paper with "try it again, better luck next time" written on it. The Fair Machine returns a candy whenever it receives a quarter as input, and returns a dime whenever a dime is input. Assume, you are puzzled by the joint behaviour of the two machines, namely by what happens whenever you connect them via the gates which are used by the Slot Machine to return the coins and by the Fair Machine to input the coins. This welding gives rise to what we call a Candy Machine; our verification system gives you a hand in understanding its behaviour.
We will provide the system with the two specifications of Slot and Fair Machines and we will get back the nondeterministic behaviour of the Candy Machine. We use the following names for the actions of our machines:
- *id*, the Slot Machine receives a dime (*input dime*);
- *od*, the Slot Machine returns a dime (*output dime*);
- *oq*, the Slot Machine returns a quarter (*output quarter*);
- *try*, the Slot Machine returns a written piece of paper (*try it again*);
- *vd*, the Fair Machine receives a dime (*vending for a dime*);
- *vq*, the Fair Machine receives a quarter (*vending for a quarter*);
- *bd*, the Fair Machine returns a dime (*back dime*);
- *bc*, the Fair Machine returns a candy (*back candy*).

```
process Candy[id, try, bc, bd] :=
        hide eq, ed in (Slot[id, try, ed, eq] |[eq, ed]| Fair[bc, bd, ed, eq])
where
        process Slot[id, try, od, oq] :=
           id; ((i; try; Slot[id, try, od, oq]) [] (i; od; Slot[id, try, od, oq])
                [] (i; oq; Slot[id, try, od, oq]))
        endproc
        process Fair[vd, vq, bc, bd] :=
            (vq; bc; Fair[vd, vq, bc, bd]) [] (vd; bd; Fair[vd, vq, bc, bd])
        endproc
endproc
```

By following the second verification strategy, we can *execute* the process Candy and derive its behaviour which can be described by means of the following action-tree:

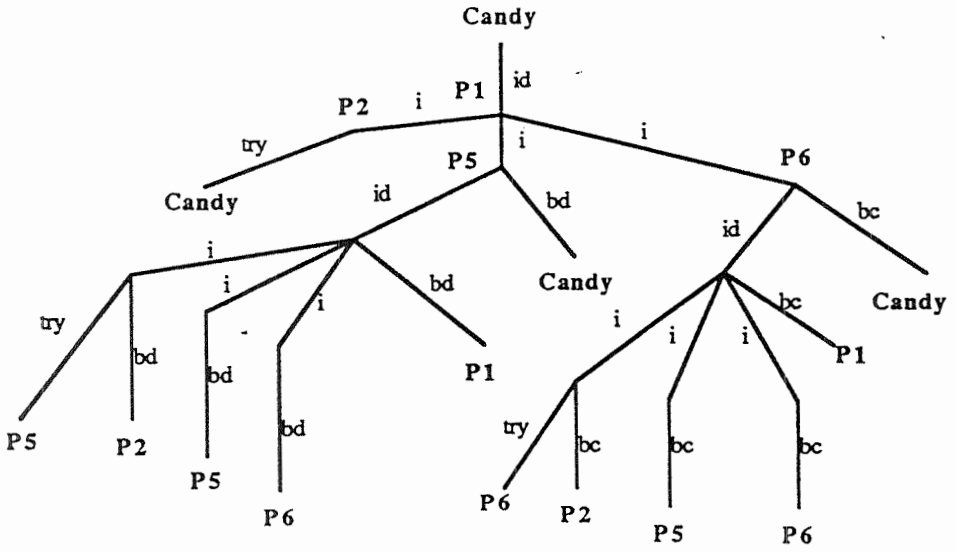

Figure 2.4.3.1 - The behaviour of a Candy Machine

Thus, Candy may be seen as a correct implementation of a specification Spec whose behaviour is the following one:

```
process Spec[id, try, bx, by] :=
        id; P1[id, try, bx, by]
where
        process P1[id, try, bx, by] :=
                (i; P2[id, try, bx, by]) [] (i; P[id, try, by, bx]) [] (i; P[id, try, bx, by])
        endproc
        process P2[id, try, bx, by] :=
                try; Spec[id, try, bx, by]
        endproc
        process P[id, try, bx, by] :=
        (id; (   (i; ((try; P[id, try, bx, by]) [] (bx; P2[id, try, bx, by])))
                 [] (i; bx; P[id, try, bx, by])
                 [] (i; bx; P[id, try, by, bx])
                 [] (bx; P1[id, try, bx, by])))
        [] (bx; Spec[id, try, bx, by])
        endproc
endproc
```

A sketch of the interaction between the user and the system is as follows:

```
?- LOTOS reduce.
/* The environment with the definitions of Candy, Slot and Fair is typed in */
Enter a process identifier: Candy
Enter a process: hide eq,ed in (Slot|[eq,ed]|Fair)

/*Omissis ... Slot and Fair are entered, then we are prompted for the process to be transformed */

Enter the process to be reduced: Candy

Trying to transform the process ...

Candy  =  hide eq,ed in (Slot|[eq,ed]|Fair)

Executing the operational semantics ...

/*Omissis ... few steps expanding terms according to the operational semantics are performed */

Candy  =  id;(i;try;hide eq,ed in (Slot|[eq,ed]|Fair)
           [](i;i;hide eq,ed in (Slot|[eq,ed]|bd;Fair)
             []i;i;hide eq,ed in (Slot|[eq,ed]|bc;Fair)))

There are reductions on Candy

Candy  =  id;(i;try;hide eq,ed in (Slot|[eq,ed]|Fair)
             [](i;hide eq,ed in (Slot|[eq,ed]|bd;Fair)
               []i;hide eq,ed in (Slot|[eq,ed]|bc;Fair)))

Testing whether some subexpressions can be replaced by process identifiers
 ....
Replaceable subexpressions have been found:

Candy = hide eq,ed in (Slot|[eq,ed]|Fair)

Folding
 ....
```

```
Candy  =  id;(i;try;Candy
            [](i;hide eq,ed in (Slot|[eq,ed]|bd;Fair)
              []i;hide eq,ed in (Slot|[eq,ed]|bc;Fair)))
```

*/*Omissis... additional steps expanding terms according to the operational semantics are performed*/*

```
Executing the operational semantics ...

Candy  =  id;(i;try;Candy
             [](i;(id;(i;(try;hide eq,ed in (Slot|[eq,ed]|bd;Fair)
                            []bd;hide eq,ed in (try;Slot|[eq,ed]|Fair))
                       .[](i;bd;hide eq,ed in (ed;Slot|[eq,ed]|Fair)
                         [](i;bd;hide eq,ed in (eq;Slot|[eq,ed]|Fair)
                         []bd;P1)))
                  []bd;Candy)
             []i;(id;(i;(try;hide eq,ed in (Slot|[eq,ed]|bc;Fair)
                            []bc;hide eq,ed in (try;Slot|[eq,ed]|Fair))
                       [](i;bc;hide eq,ed in (ed;Slot|[eq,ed]|Fair)
                       [](i;bc;hide eq,ed in (eq;Slot|[eq,ed]|Fair)
                       []bc;P1)))
                 []bc;Candy)))

There are no reductions on Candy
Testing whether some subexpressions can be replaced by process identifiers
 ....
Replaceable subexpressions have been found:

P5 = hide eq,ed in (Slot|[eq,ed]|bd;Fair)
P2 = hide eq,ed in (try;Slot|[eq,ed]|Fair)
P5 = hide eq,ed in (ed;Slot|[eq,ed]|Fair)
P6 = hide eq,ed in (eq;Slot|[eq,ed]|Fair)
P6 = hide eq,ed in (Slot|[eq,ed]|bc;Fair)

Folding
 ....
Candy  =  id;(i;try;Candy
            [](i;(id;(i;(try;P5[]bd;P2)[](i;bd;P5[](i;bd;P6[]bd;P1)))
                 []bd;Candy)
            []i;(id;(i;(try;P6[]bc;P2)[](i;bc;P5[](i;bc;P6[]bc;P1)))
                 []bc;Candy)))

Executing the operational semantics ...
```

/ Only previously computed transitions can be executed */*

The reduced form of the input process is the following:

*/*Omissis ... the above reduced form of Candy is printed with the bindings of identifiers P1 ... P6*/*

This concludes our session. An alternative way of looking at this example is to prove that the original specification of the parallel composition of Slot and Fair Machines is observational equivalent to the behaviour described by Figure 2.4.3.1.

2.5 Existing or Potential Tool Support

Besides the tool described in section 2.4, there are already a few environments which support verification of concurrent systems properties: Concurrency Workbench [CPS88], TAV [GLZ89], Auto [V86] and Squiggles [BC89]. They can be used to verify behavioural equivalences of specifications and to decide whether a specification satisfies a logical (modal) property. In performing verification, most of these systems follow an approach which is based on a finite state machine representation of terms and makes use of a generalized partitioning algorithm. The only exception is the TAV system, directly based on the definition of bisimulation.

Verification systems based on finite state machines have two main disadvantages.

First, when verifying equivalence of two given specifications, most of the time these systems deliver a yes/no answer and the answer is often no. In this case the user has not sufficient control over the verification process to get suggestions about what went wrong and where, within the specification, the error is located.

Second, when dealing with interleaving descriptions of concurrent processes these systems lead very quickly to state explosion and the user has no way of controlling the flow of the proof which would allow him to prune the state space.

To this respect the tool described in section 2.4 follows a different approach that, on one side, tries to overcome some of those disadvantages and, on the other side, proposes an open ended environment in which the user can define his own strategies.

References

[Bau 85] Bauer et al: The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L. *LNCS 183*, Springer (1985)

[Bau 87] Bauer et al: The Munich Project CIP. Volume II: The Program Transformation System CIP-S. *LNCS 292*, Springer (1987)

[BB 87] T. Bolognesi, E. Brinksma: Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems 14*, (1987), 25-59.

[BD 77] R. M. Burstall; J. Darlington: A Transformation System for Developing Recursive Programs. *Journal of the Association of Computing Machinery, Vol. 24, No. 1*, (1977) 44-67

[Br 88] E. Brinksma (ed.): Information processing systems – open systems interconnection – LOTOS. A formal description technique based on the temporal ordering of observational behaviour. *International Standard*, ISO 8807

[BW 84] F. L. Bauer; H. Wössner: Algorithmische Sprache und Programmentwicklung. Springer (1984)

[BC 89] T. Bolognesi, M. Caneve: Squiggles - A Tool for the Analysis of LOTOS Specifications. *in Formal Description Techniques (K. Turner, ed.)*, North-Holland, (1989), 201-216

[BW 87] L. G. Bouma, H. R. Walters: Implementing Algebraic Specifications. *Report P8714*, University of Amsterdam, Department of Mathematics and Computer Science, Programming Research Group (1987)

[Cl 88] Ingo Claßen: Algebraische Grundlagen der Termersetzung mit bedingten Gleichungen. *Bericht-Nr. 88-04, TU Berlin, Fachbereich Informatik*, (1988).

[CPS 88] R. Cleaveland, J. Parrow, B. Steffen: The Concurrency Workbench: Operating Instructions. *Tech. Note 2/88*, University of Sussex, (October 1988)

[Ehr 89] H. Ehrig: Concepts and Compatibility Requirements for Implementations and Transformations of Specifications. *EATCS-Bulletin, Algebraic Specification Column Part 6*, July 1989, see also LOTOSPHERE working paper *Lo/WP1/T1.2/TUB/N0003*

[EKMP 82] H. Ehrig; H.-J. Kreowski; B. Mahr; P. Padawitz: Algebraic Implementation of Abstract Data Types. *Theoretical Computer Science 20*, (1982) 209-263

[EK 83] H. Ehrig; H.-J. Kreowski:Compatibility of Parameter Passing and Implementation of Parameterized Data Types. *Theoretical Computer Science 27*, (1983) 255-286

[DH 84] R. De Nicola, M. Hennessy: Testing Equivalences for Processes. *TCS 34*, (1984), 83-133.

[DIN 89] R. De Nicola, P. Inverardi, M. Nesi: Using the Axiomatic Presentation of Behavioural Equivalences for Manipulating CCS Specifications. *to appear in Proc. Workshop on Automatic Verification Methods for Finite State Systems, LNCS*, Springer Verlag, (1989).

[FH 88] A.J. Field; P.G. Harrison: Functional Programming. *International Computer Science Series*, Addison Wesley (1988)

[Ga 88] H. Ganzinger: A Completion procedure for conditional equations. *LNCS 308, Proc. 1st Int'l Workshop on Conditional Term Rewriting, Orsay 1987* Springer (1988).

[GLZ 89] J.C. Godesken, K.G. Larsen, M. Zeeberg: TAV Users Manual. *Internal Report*, Aalborg University Center, Denmark, (1989)

[HM 85] M. Hennessy, R. Milner: Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM, 32, No. 1*, (1985), 137-161.

[HL 79] G. Huet, J.-J. Levy: Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems. *Technical Report 359*, INRIA (1979).

[HO 82] C. Hoffmann, M. O'Donnell: Pattern matching in trees. *Journal of the ACM, 68-95* (1982).

[IN89] P. Inverardi, M. Nesi: A Rewriting Strategy to Verify Observational Congruence. *I.E.I. Internal Report Nr. B4-38*, Pisa, (August 1989)

[KB 70] D. Knuth, P. Bendix: Simple Word Problems in Universal Algebras. *in J. Leech, ed., Computational Problems in Abstract Algebra* Pergamon Press, Oxford, UK (1970).

[Kl 87] Jan W. Kloop.: Term Rewriting Systems: A Tutorial. *EATCS Bulletin No. 32* (June 1987).

[NO 86] M. Navarro and F. Orejas: Proof Rules for Conditional Equations. *Res. Rep., Facultat d'Informatica de Barcelona*, (1986).

[Pad 88] P.Padawitz: Computing in Horn Clause Theories. *Springer EATCS Monographs on Theor. Com. Sci.*, 1988

[Sch 88] M. Schmitz: Bedingte Termersetzungssysteme. *Interner Bericht* Fachbereich Informatik, Universität Kaiserslautern, Projektarbeit SS 88 (1988).

[ST 88] D. Sannella; A. Tarlecki: Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited. *Acta Informatica 25*, (1988) 233-281

[St 89] R. Strandh: Classes of Equational Programs that Compile into Efficient Machine Code. *LNCS 355, Proc. Rewriting Techniques and Applications, 3rd International Conference, Chapel Hill, USA*, Springer (1989).

[VG 86] D. Vergammi: Verification bz means of observational equivalence on automata. *Rapports de Recherche, INRIA Nr. 501*, INRIA, (1986)

[WB 89] D. Wolz, P.Boehm: Compilation of LOTOS Data Type Specifications. *Proc. of the IFIP TC6-WG 6.1, 9th International Symposium on Protocol Specification, Testing and Verification (ed. E.Brinksma,G.Scollo,C.A.Vissers)* ,1989

[Wo 89] D. Wolz: Theoretische Grundlagen der Compilation algebraischer Spezifikationen und Termersetzungssysteme. *Bericht-Nr. 89-xx*, Technische Universität Berlin, Fachbereich 20 (1989).