TECHNION - Israel Institute of Technology
Computer Science Department

A PROGRAM TRANSFORMATION REGARDED AS A PROOF TRANSFORMATION

by

L. Shrira and N. Francez

Technical Report #371

July 1985

A PROGRAM TRANSFORMATION REGARDED AS A PROOF TRANSFORMATION

by

Liuba Shrira and Nissim Francez
Computer Science Dept.
The Technion, Haifa, ISRAEL

1. Introduction

In [SFR83] a program transformation was presented, transforming recursive sequential programs into recursive distributed programs, provided that distributed implementations are given for the sequential "pieces" between recursive calls. The use of such a transformation was suggested there as a good methodology for obtaining distributed programs in view of the non-trivial results obtained by applying the transformation to several k-selection programs, obtaining new distributed k-selection algorithms.

The purpose of this paper is to view the above mentioned transformation as a *proof transformation*: given a correctness proof for the source sequential program, derive systematically a correctness proof for the distributed program resulting by applying the program transformation. We regard the proof transformation as an interesting exercise in program verification and as an indirect proof of the correctness of the program transformation itself.

In order to carry out such a task, the discussion has to be linked to *concrete* programming languages for which a deductive proof-system is known. For the sequential programming language we take a simple language with parameterless recursion, similar to the standard languages considered in the context of formal semantics [deB80]. As for the language for distributed programming, we chose a variant of **CSP** [H78], both because our familiarity with the language and because of the proof system available for it [AFR80]. We use here Apt's new formulation of it [A84], using the *proof assumptions* paradigm.

As the original **CSP** had no procedures mechanism, we define an extension, called **CSPR**, having local parameterless recursive procedures. The extension to parametrized recursion poses no further problems.

Some acquaintance with **CSP** and the proof system in [AFR80] is needed in

order to understand the paper.

2. Notations and definitions

Define the programming language **PL** to include: *skip*, assignments, sequential composition, guarded commands, parameterless recursive procedure declarations (without nested declarations, for simplicity) and procedure *call*s.

Next, define **CSPR** as **CSP** augmented with *local* parameterless recursive procedures. This is a rather modest extension, where we have *recursion within concurrency*, as opposed to *recursion across concurrency*, e.g [LBCL82], where recursive calls may spawn new processes, creating dynamic networks.

In both languages, procedure declarations are denoted by R <- *proc* BR *end*, where R is the procedure name and BR its body. The keyword *call* is used for procedure calls.

As the assertion language (for both programming languages) we take **L**, a first order language with equality. We use a, b, x, y, z to denote variables of L, s, t to denote terms (expressions) of L, e to denote a quantifier-free formula of L and p, q to denote the formulae (assertions) of L.

A proof system for partial correctness assertions for **PL** is standard (see [AB1]). A proof system for partial correctness assertions for **CSPR** is obtained from the corresponding rules for **CSP** [AFR80, A84] by the additional rule for (local) recursion appearing in separate process proofs (see figure 1).

Explanations: soundness and completeness of the local recursion rule follow from soundness and completeness of the recursion rule in the sequential context. An assertion $\{p_\alpha\}\ \alpha\ \{q_\alpha\}$ about a communication command α in the body of a recursive procedure joins the i/o-assumptions set and is discharged after the *cooperation proof* [AFR80, A84]. At execution time such a command α may be

$$\frac{A, \{p\}\ R\ \{q\}\ \vdash\ \{p\}\ BR\ \{q\}}{A \vdash\ \{p\}\ R\ \{q\}}$$

Figure 1. The (local) recursion rule

executed several times. The cooperation proof guaranties that if it is executed then its post condition q_a is established. If for some instance of a no matching i/o command is available (leading to deadlock or failure) then a is not executed - without contradiction to the partial correctness assertion. An example of application of the local recursion rule in a partial correctness proof of a simple **CSPR** program appears in the appendix.

3. The proof transformer

Assume that a sequential **PL** program S (see figure 2.) and a proof of $\{p\}S\{q\}$ are given. The assertions p and q may refer to all the variables in S. Note that p, q are the respective pre - and post - conditions associated with the (unique) recursive call to R. NR^0 denotes the *exit* part (containing no recursive calls), e denotes the corresponding *exit* condition and NR^1, NR^2 denote the nonrecursive parts of S. Handling more than one recursive call (and more than two nonrecursive parts) is similar. Assume a given fixed network with nodes N_1 ... N_n. We want to assign process P_i to node N_i. Let $V=free(p) \cup free(q)$. A *distribution* **V** of V is a partition of V into $\{V_i\}$ $i=1...n$; V_i is taken as a local state for P_i. Assume the nonrecursive parts of R, namely NR^0, NR^1, NR^2 and the computation of the *exit condition* e have (given) distributed implementations in **CSPR** with respect to distribution **V**: $DNR^k, k=0,1,2$ and E, all on the same given network. Also given are the correctness proofs of these distributed implementations. These implementations

---

S:: $\{p\}$ R <-- *proc*

[$e \rightarrow NR^0$

[]

$\neg e \rightarrow NR^1$;

$\{p\}$ *call* R $\{q\}$

NR^2]

end_R $\{q\}$;

$\{p\}$ *call* R $\{q\}$

Figure 2. The sequential LP program S

should satisfy the following correctness criteria (see figure 3), where I^e, I^0, I^1, I^2 are some global invariants (GI's) establishing *cooperation* (as required by the CSP proof system) and; without loss of generality (due to the appropriate substitution rules in the CSP proof system), we assume that $p \equiv \Lambda p_i$, $free(p_i) \subseteq V_i$ and $q \equiv \Lambda q_i$, $free(q_i) \subseteq V_i$ and $e \triangleq de_i$, $free(de_i) \subseteq V_i$.

We construct DS, a complete distributed implementation for S, together with its correctness proof, out of E, DNR^0, DNR^1 and DNR^2, again using the same given network. DS should satisfy $\{ I^* \Lambda p \} [DS] \{ I^* \Lambda q \}$ with respect to some GI I^* establishing cooperation. Take DS to be as shown in figure 4. We do not explicitly define the code for the synchronization operation $synch_i^k$ at this stage. Rather, in the process of proof construction, sufficient conditions for *synch* are derived, guaranteeing the validity of the constructed proof. These conditions have to be met by any correct implementation of *synch*. The main function of *synch* is to *synchronize* the independent recursive calls in the separate processes, to form a correct distributed implementation of the global recursion in S, as suggested in the transformation in [SFR83].

The last steps in proving
$\{ p \} E \{ p \Lambda \Lambda_{i=1,n} (de_i \equiv e) \}$, $\{ p \Lambda e \} DNR^0 \{ q \}$,
$\{ p \Lambda \neg e \} DNR^1 \{ p \}$, and $\{ q \} DNR^2 \{ q \}$ are the applications of the parallel composition rule [AFR 80], instantiated as in the figure 5. The last step in the proof of DS can be taken to be the application of the parallel composition rule instantiated as in the figure 6.

In this application, $A_i^* = A_i^e \cup A_i^0 \cup A_i^1 \cup A_i^2 \cup A_i^{synch^e} \cup A_i^{synch^0} \cup A_i^{synch^1} \cup A_i^{synch^2}$, $p = \Lambda_{i=1,n} p_i$, $q = \Lambda_{i=1,n} q_i$ and $A_i^{synch^j}$ is the i/o assumption set for $synch_i^j$, $j = e, 0, 1, 2$ respectively.

$$\{ I^e \Lambda p \} E :: [E_1 \| \cdots \| E_n] \{ I^e \Lambda p \Lambda \Lambda_{i=1,n} (de_i \equiv e) \}$$

$$\{ I^0 \Lambda p \Lambda e \} DNR^0 :: [DNR_1^0 \| \cdots \| DNR_n^0] \{ I^0 \Lambda q \}$$

$$\{ I^1 \Lambda p \Lambda \neg e \} DNR^1 :: [DNR_1^1 \| \cdots \| DNR_n^1] \{ I^1 \Lambda p \}$$

$$\{ I^2 \Lambda q \} DNR^2 :: [DNR_1^2 \| \cdots \| DNR_n^2] \{ I^2 \Lambda q \}$$

Figure 3. Correctness criteria for distributed implementations of nonrecursive parts

$DS :: [DS_1 \| \cdots \| DS_n]$

where

$\{p_i\}\, DS_i ::$

$DR_i \leftarrow\!\!- proc$

$synch_i^0;$

$E_i;$

$[de_i \rightarrow synch_i^0; \{p_i \ \Lambda \ de_i\}\ DNR_i^0\ \{q_i\}$

$[]$

$\neg de_i \rightarrow synch_i^1;$

$\{p_i \ \Lambda \ \neg de_i\}\ DNR_i^1\ \{p_i\};$

$call\ DR_i;$

$synch_i^2;$

$\{q_i\}\ DNR_i^2\ \{q_i\}$

end_DR_i

$call\ DR_i\ \{q_i\}$

Figure 4. The distributed program DS

$$\frac{A_i^e \vdash \{ p_i \}\ E_i\ \{ p_i \},\ i=1...n \quad A_i^e \text{ cooperate w.r.t. } I^e,\ i=1...n}{\{ p \}\ E\ \{ p \}} \quad e)$$

$$\frac{A_i^0 \vdash \{ p_i \wedge de_i \}\ DNR_i^0\ \{ q_i \},\ i=1...n \quad A_i^0 \text{ cooperate w.r.t. } I^0,\ i=1...n}{\{ p \wedge e \}\ DNR^0\ \{ q \}} \quad 0)$$

$$\frac{A_i^1 \vdash \{ p_i \wedge \neg de_i \}\ DNR_i^1\ \{ p_i \},\ i=1...n \quad A_i^1 \text{ cooperate w.r.t. } I^1,\ i=1...n}{\{ p \wedge \neg e \}\ DNR^1\ \{ p \}} \quad 1)$$

$$\frac{A_i^2 \vdash \{ q_i \}\ DNR_i^2\ \{ q_i \},\ i=1...n \quad A_i^2 \text{ cooperate w.r.t. } I^2,\ i=1...n}{\{ q \}\ DNR^2\ \{ q \}} \quad 2)$$

Figure 5. Applications of parallel composition rule DNR^j

$$3)\ \frac{A_i^* \vdash \{ p_i \}\ DS_i\ \{ q_i \},\ i=1..n \quad A_i^* \text{ cooperate w.r.t. } I^*,\ i=1...n}{\{ I^* \wedge p \}\ DS\ \{ I^* \wedge q \}}$$

Figure 6. Application of parallel composition rule to DS

In order to apply 3) it is needed to prove the assertions $A_i^* \vdash \{ p_i \}\ DS_i\ \{ q_i \}$. Each DS_i is a call to a recursive procedure DR_i which in turn is proved by application of the following instance of the local recursion rule (see figure 7). The proof of the premises of 4) goes as follows:

1. $\{ p_i \}\ synch_i^s\ \{ p_i \},\quad vars(p_i) \cap vars(synch_i^s) = \phi$

$$4)\ A_i^*,\ \{\ p_i\ \}\ call\ DR_i\ \{q_i\ \}\ |-\ \{\ p_i\ \}\ BDR_i\ \{\ q_i\ \}$$

$$A_i^*\ |-\ \{\ p_i\ \}\ DR_i\ \{\ q_i\ \}$$

Figure 7. Applications of the local recursion rule

2. $A_i^e\ |-\ \{\ p_i\ \}\ E_i\ \{\ p_i\ \}$, *by* e).

3. $\{\ p_i\ \wedge\ de_i\ \}\ synch_i^0\ \{\ p_i\ \wedge\ de_i\ \}$, $vars(p_i) \cap vars(synch_i^0)=\phi$

4. $A_i^0\ |-\ \{\ p_i\ \wedge\ de_i\ \}\ DNR_i^0\ \{\ q_i\ \}$, *by* 0).

5. $\{\ p_i\ \wedge\ \neg de_i\ \}\ synch_i^1\ \{\ p_i\ \wedge\ \neg de_i\ \}$, $vars(p_i) \cap vars(synch_i^1)=\phi$

6. $A_i^1\ |-\ \{\ p_i\ \wedge\ \neg de_i\ \}\ DNR_i^1\ \{\ p_i\ \}$, *by* 1).

7. $A_i^*\ |-\ \{\ p_i\ \}$ *call* $DR_i\ \{\ q_i\ \}$; *by recursion assumption.*

8. $\{\ q_i\ \}\ synch_i^2\ \{\ q_i\}$, $vars(q_i) \cap vars(synch_i^2)=\phi$

9. $A_i^2\ |-\ \{\ q_i\ \}\ DNR_i^2\{\ q_i\ \}$, *by* 2)

10. $A_i^*\ |-\ \{\ p_i\ \}\ BDR_i\ \{\ q_i\ \}$.

where $A_i^* = \cup_{j\ \in\ \{e,0,1,2\}} A_i^j \cup \cup_{j\ \in\ \{e,0,1,2\}} A_i^{synch^j}$.

To complete the premises of 3) cooperation has to be proved for A^*. We want to find such a global invariant that allows us to make use of the cooperation proofs from e), 0), 1) and 2).

Consider all the syntactically matching i/o pairs arising in A^*. Call "new" those pairs not handled in e), 0), 1), 2) and the cooperation proofs for $synch^0$, $synch^1$, $synch^2$. In the construction of $synch$, we require that no new pairs arise from assumptions in A^e, A^0, A^1 and A^2 with assumptions in A^{synch^j}, $j \in \{0,1,2\}$ (by means of tagged messages). The only new pairs may originate from assumptions about i/o commands in

1. A^e, A^0, A^1 and A^2 or

2. i/o commands inside A^1 active at different recursion levels - thus not handled in 1). These pairs have passed cooperation vacuously with I^1 (and

similarly for A^e, A^0 and A^2).

It is *synch* that prevents those communications from actually happening. Thus to handle these pairs in the proof we introduce a derived global invariant I^*, that by expressing the effect of *synch* ensures vacuous cooperation for the "new" pairs. In addition I^* has to positively simulate I^e, I^0, I^1 and I^2 for communications that do occur.

Note: if no "new" pairs arise in A^* - no *synch* is required. I.e. no matching pairs arise from $\cup_{j \in \{e,0,1,2\}} A^j$ and no i/o pairs passed cooperation vacuously inside $A^j, j \in \{e,0,1,2\}$. A careful analysis of this situation may lead to optimize the program transformation accordingly, avoiding excessive synchronization. (See the simple example in appendix).

In the next section a specific *synch* implementation is suggested and a global invariant I^* is presented (see figure 8). The effect of *synch* is expressed by introducing auxiliary variables at_i, $atsynch_i$ and l_i. at_i indicates the currently executing nonrecursive part, $atsynch_i$ indicates that the execution is in the synchronizing part, and l_i keeps track of the recursion level.

4. Synch implementation

Synch is implemented as a *closed layer* [EF82] preserving the global invariant I^*. As any execution of a **CSP** program with a closed layer is equivalent to the execution where all processes are delayed before entering the closed layer and

$$I^* \equiv \bigwedge_{i=1,n} \neg atsynch_i \rightarrow$$

$$\left(\bigwedge_{i=1,n} at_i=e \;\vee\; \bigwedge_{i=1,n} at_i=0 \;\vee\; \bigwedge_{i=1,n} at_i=1 \;\vee\; \bigwedge_{i=1,n} at_i=2\right) \wedge\; l_1=\ldots=l_n$$

$$\wedge\; \bigwedge_{i=1,n} at_i=e \rightarrow I^e$$

$$\wedge\; \bigwedge_{i=1,n} at_i=0 \rightarrow I^0$$

$$\wedge\; \bigwedge_{i=1,n} at_i=1 \rightarrow I^1$$

$$\wedge\; \bigwedge_{i=1,n} at_i=2 \rightarrow I^2$$

Figure 8. The derived global invariant I^*

then enter and exit the layer together (by [EF82]) this is sufficient for our needs. We present below a possible implementation for $synch^e$ in **CSP** (see figure 9). It makes use of some spanning tree **T** over the N_i -s. The synchronizing signals are collected up the tree (*up*) and transmitted back down the tree (*down*). For $synch^0$ and $synch^1$ the only difference is in the assignment: at_i:=0 and at_i:=1 , respectively, and in that the level variable l need not be updated. For $synch^2$ the difference is in the assignments: $at_i := 2$, $l_i := l_i - 1$;
For leaf and root nodes the programs are changed correspondingly.

Note: Because of the restriction on the structure of bracketed sections in the **CSP** proof system, the above described sections are to be splitted to contain one communication per bracketed section.

What remains to be checked is the preservation of the invariants I^e, I^0, I^1 and I^2 by the corresponding bodies of E, DNR^0 and DNR^1. By the completeness theorem for the proof system of **CSP** it is always possible to find global invariants containing only auxiliary variables. Thus by simple auxiliary variable renaming the variable clashes between DNR^j and I^k are prevented.

All the above applies as well to any pattern of simple recursion.

The program at some internal node in T

≪ $atsynch_i$:=*true*;

at_i:=e;l_i:=l_i+1;

receive up() from all children ≫;

send up() to parent;

receive down() from parent;

≪ *send down() to all children*; $atsynch_i$:=*false* ≫

Figure 9. An implementation of $synch^e$

Discussion and future directions

A natural question arising in the context of above is the extension of the proof transformer to handle total correctness proofs. More specifically the question is: how given a sequential recursive program and its total correctness proof and distributed implementations for the nonrecursive parts with their total correctness proofs, to construct a total correctness proof for the whole transformed program. Unfortunately, we have no answer to this question- the main problem being the following: when using the total correctness proof system [A83] one has to construct termination proofs for each sequential process and then establish cooperation. The termination of the local recursive procedures follows from the termination of the sequential recursive program, but it is not clear how to make use of the termination proof of the sequential procedure in the termination proofs for the local processes.

Another interesting extension follows from the previously made observation that examination of the proof transformer may suggest an algorithmic optimization with respect to given distributed implementations of nonrecursive parts, i.e. by proving that no new semantic matches arise in the complete DS, the synch part is proven redundant. In a forthcoming paper [GS85] a more general criteria is suggested. Here we view a distributed implementation of each DNR^i as a stand alone closed layer and suggest a proof rule for proving such closeness. Synch is redundant for any composition of stand alone closed layers.

Acknowledgements

We would like to thank Krzysztof Apt and Orna Grumberg for helpful discussions.

References

[AB1]
K.R. Apt, "Ten Years of Hoare's Logic: A Survey-Part I", ACM-TOPLAS 3, No. 4, pp. 431-483, 1981.

[AB4]
K.R. Apt, "Proving Correctness of CSP Programs - A Tutorial" *Proc. International Summer School "Control Flow and Data Flow: Consepts of Distributed Programming"*, *Marktoberdorf*, *Springer-Verlag*, to appear.

[AFR80]
K.R. Apt, N. Francez, W.P. de Roever, "A proof system for communicating sequential processes", *ACM-TOPLAS* 2, No. 3, pp. 359-380, 1980.

[LBCL82]
D. Lehmann, A. Barak, S. Cohen, J. Levy, "Communicating Processes: A Language for Dynamic Concurrent Systems", Institute of Mathematics and Computer Science, Hebrew University, 1982.

[deB80]
J.W. de Bakker, **MAthematical Theory of Program Correctness**, *Prentice Hall*, 1980.

[EF82]
T. Elrad, N. Francez, "Decomposition of distributed programs into communication-closed layers" ,*Science of Computer Programming* No.2, pp. 155-173, 1982.

[H78]
C.A.R. Hoare, "Communicating Sequential Processes", *Comm. ACM* 21, No. 8, pp. 666-677, 1978.

[SFR83]
L. Shrira, N. Francez, M. Rodeh, "Distributed k-selection: from a sequential to a distributed algorithm", *Proceedings of 2nd ACM PODC*, *Montreal*, 1983.

Appendix Deriving a correctness proof for a **CSPR** factorial program.

We consider a distributed implementation of a factorial procedure *Factorial* on a three processor net: one of the processors (P1) is able to multiply, one (P2) to subtract and the third (P3) to add. To derive the correctness proof of the **CSPR** implementation of *Factorial* we make use of a sequential procedure proof and the proofs of the distributed **CSPR** implementations for the nonrecursive parts. Consider the annotated sequential factorial procedure below where consecutive occurrences of assertions $\{ p \}\{ q \}$ imply $p \rightarrow q$ and appeal to the *consequence* rule.

Factorial <— *proc*

$\{ x \geq 0 \}\ [\, x=0 \rightarrow \{ x=0 \}\ y:=1\ \{ y=x! \}$

$[]$

$x \neq 0 \rightarrow \{ x-1 \geq 0 \}\ x:=x-1;\ \{ x \geq 0 \}$

$\{ x \geq 0 \}$ *call Factorial*; $\{ y=x! \}$

$\{ y \cdot (x+1)=(x+1)! \}\ x:=x+1\ \{ y \cdot x=x! \}\ y:=y \cdot x\ \{ y=x! \}\ ]$

end_Factorial

The non recursive parts here are

$e :: x=0$

$NR^0 :: y:=1$

$NR^1 :: x:=x-1;$

$NR^2 :: x:=x+1;\ y:=y \cdot x;$

Assume the following annotated distributed **CSP** implementations for the nonrecursive parts, with E_i, DNR_i^0, DNR_i^1, DNR_i^2 assigned to P_i $i=1,2,3$. For convenience, we omit in the texts the *pre-* and *post-* conditions equal to *true* .

$E::\ \{ x \geq 0 \}$

$[\, E_1 :: \ll P_2!x >>;\ \ll P_3!x \gg \,] \parallel E_2 :: \ll P_1?n \gg \parallel E_3 :: \ll P_1?m \gg \,]$

$\{ x \geq 0 \ \wedge\ (n=0 \equiv x=0 \ \wedge\ m=0 \equiv x=0) \}$

establishing the global invariant : $I^0 \equiv n=x \ \wedge\ m=x$

$DNR^0 : \{ x \geq 0 \ \wedge\ x=0 \}$

$[\, DNR_1^0 :: y:=1 \parallel DNR_2^0 :: skip \parallel DNR_3^0 :: skip \,]\ \{ y=x! \}$

$DNR^1 : \quad \{ x \geq 0 \ \wedge\ x=0 \}$

$[\, DNR_1^1 :: [\, \ll P_2!x >>;\ \ll P_2?x \gg \,]$

$\parallel DNR_2^1 :: [\, \ll P_1?n >>;\ \ll n:=n-1; P_1!n \gg \,] \parallel DNR_3^1 :: skip \,]\ \{ x \geq 0 \}$

with global invariant : $I^1 \equiv n=x$

DNR^2 :: { $y \cdot (x+1)=(x+1)!$ }

[DNR_1^2 :: [$P_2!x$; $P_2?x$; $y:=y \cdot x$] || DNR_2^2 :: *skip*

|| DNR_3^2 :: [$P_1?m$; $m:=m+1$; $P_1!m$]] { $y=x!$ }

with global invariant: $I^2 \equiv m=x$

We construct the following complete annotated program with DS_i assigned to processor P_i, $i=1,2,3$.

(1): DS: { $x \geq 0$ } [DS_1 || DS_2 || DS_3] { $y=x!$ } .

where

DS_1 :: DR_1 <-- *proc*

$synch_1^e$;

$\ll P_2!x \gg$; $\ll P_3!x \gg$;

[$x=0 \rightarrow synch_1^0$; $y:=1$ { $y=x!$ }

[]

$x \neq 0 \rightarrow synch_1^1$; { $x>0$ } $\ll P_2!x \gg$

{ $x>0$ } $\ll P_2?x \gg$ { $x \geq 0$ } *call* DR_1; { $y=x!$ }

$synch_1^2$;

{ $y \cdot (x+1)=(x+1)!$ } $\ll P_3!x \gg$;

{ $y \cdot (x+1)=(x+1)!$ } $\ll P_3?x \gg$; { $y \cdot x=x!$ } $y:=y \cdot x$ { $y=x!$ }]

end DR_1;

{ $x \geq 0$ } *call* DR_1 { $y=x!$ }

DS_2 :: DR_2 <-- *proc*

$synch_2^e$;

$\ll P_1?n \gg$;

[$n=0 \rightarrow synch_2^0$; *skip*

[]

$n \neq 0 \rightarrow synch_2^1$;

$\ll P_1?n \gg$;

$\ll n:=n-1$; $P_1!n \gg$; *call* DR_2; $synch_2^2$;

end DR_2

call DR_2

DS_3 :: DR_3 <— *proc*

synch$_3^2$;

≪ $P_1?m$ ≫;

[$m=0$ → *synch*$_3^0$; *skip*

[]

$m\neq 0$ → *synch*$_3^1$; *call* DR_3; *synch*$_3^2$;

≪ $P_1?m$ ≫;

≪ $m:=m+1$; $P_1!m$ ≫;

end DR_3

call DR_3

To prove (1), by the parallel composition rule we need to prove:

(2): A_1 |– { $x\geq 0$ } DS_1 { $y=x!$ }

(3): A_2 |– { *true* } DS_2 { *true* }

(4): A_3 |– { *true* } DS_3 { *true* }

For (3),(4) the proofs are immediate.

For DS_1, (i.e. (2)) the procedure call is to be proved by:

(5): { $x\geq 0$ } *call* DR_1 { $y=x!$ }

To prove (5) we prove the locally recursive DR_1 procedure using the local recursion rule introduced above. By this rule we need to prove:

(6): { $x\geq 0$ } *call* DR_1 { $y=x!$ } |– { $x\geq 0$ } BDR_1 { $y=x!$ }

The proof outline for (6) was already presented with the program itself.

To establish the cooperation for (1) the following instantiation of the global invariant I^* is used:

$I^* \equiv \Lambda_{i=1,3}$ ¬*atsynch*$_i$ →

$(\Lambda_{i=1,3} at_i=e \vee \Lambda_{i=1,3} at_i=0 \vee \Lambda_{i=1,3} at_i=1 \vee \Lambda_{i=1,3} at_i=2) \wedge l_1=l_2=l_3)$

$\Lambda \Lambda_{i=1,3} at_i=e \rightarrow n = x \wedge m=x$

$\Lambda \Lambda_{i=1,3} at_i=1 \rightarrow n = x \wedge \Lambda \Lambda_{i=1,3} at_i=2 \rightarrow m = x$

Now cooperation needs to be established for the i/o assumptions . The following pairs were already handled in the cooperation proofs for E, DNR^0, DNR^1 and DNR^2. As with the help of the auxiliary variables handled by *synch* the corresponding invariants I^j,$j \in \{e,1,2\}$ are enabled by the I^* , the proof for them is complete. Other pairs, by I^*, pass cooperation vacuously.

1. { *true* $\wedge$ I^* } ≪ $P_2!x$ ≫ || ≪ $P_1?n$ ≫ { *true* $\wedge$ I^* }

2. $\{ \mathit{true} \ \Lambda \ I^* \} \ll P_3!x \gg \| \ll P_1?m \gg \{ \mathit{true} \ \Lambda \ I^* \}$

3. $\{ x>0 \ \Lambda \ I^* \} \ll P_2!x \gg \| \ll P_1?n \gg \{ x>0 \ \Lambda \ I^* \}$

4. $\{ x>0 \ \Lambda \ I^* \} \ll P_2?x \gg \| \ll n:=n-1; \ P_1!n \gg \{ x \geq 0 \ \Lambda \ I^* \}$

5. $\{ y \cdot (x+1)=(x+1)! \ \Lambda \ I^* \} \ll P_3!x \gg \| \ll P_1?m \gg \{ y \cdot (x+1)=(x+1)! \ \Lambda \ I^* \}$

6. $\{ y \cdot (x+1)=(x+1)! \ \Lambda \ I^* \} \ll P_3?x \gg \| \ll m:=m+1; \ P_1!m \gg \{ y \cdot x=x! \ \Lambda \ I^* \}$

Note: as no actual "new" pairs arise in A^*, the *synch* part is redundant in this simple example and leads to excessive synchronization.