Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

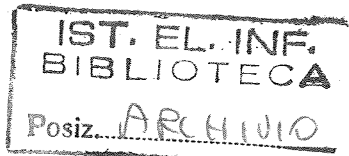
PISA

Expliciting Dynamic Process Generation

Chapter 9 of
"Catalogue of LOTOS Correctness Preserving Transformations"
ESPRIT Project 2304 LOTOSPHERE
task 1.2 Third Deliverable

A. Fantechi, S. Gnesi

Nota Interna B4-30
Luglio 1992



Chapter 9

Expliciting Dynamic Process Generation

9.1 Informal Description

Consider the following informal specification: Whenever a request is made on the gate g, instantiate a new process that handles such request, so that other requests could be served in parallel. This paradigm can be frequently found in several classes of application. An example can be found in the Telephone System specification in [19]. Dynamic instantiation of new processes can be expressed in LOTOS by means of an unguarded recursion (see the example below). The proposed transformation, starting from a process P, which contains unguarded recursions, produces a process Q with only guarded recursions, which is observationally equivalent to P.

9.2 Motivation

Unguarded recursion is admitted in LOTOS, but still can give problems with some tools operating on the language; for example, in [29] a LOTOS interpreter is described which is not able to execute some kinds of unguarded recursions. This is due to the attempt made by the interpreter (in the case of unguarded recursion used for process generation) to create new processes endlessly, before performing any of their first actions. On the other hand, the use of unguarded recursion for modelling dynamic generation of processes can be done at an abstract level of specification, but gives little information on how to implement it. The proposed transformation gives an hint in this direction, by expliciting the action of "creating a new process".

9.3 Example

The following specification describes a Server which accepts requests and which handles them independently and concurrently, operating on a central repository of information; each request activates a new instance of the handler process for that request. Unguarded recursion is used to specify activation of new instances. Two equivalent specifications with guarded recursion are given in section 9.6, obtained with the two solutions proposed.

```
specification ClientServer[Req, Reply] : noexit :=

        (* Abstract Data Types Definitions*)

behaviour
hide int_gate in
(Handlers [Req, Reply, int_gate]
| [int_gate] |
Central_repository [int_gate])

where
process  Handlers [Req, Reply, int_gate]  noexit :=
Handlers [Req, Reply, int_gate]
|||
Handler [Req, Reply, int_gate]

where
process  Handler [Req, Reply, int_gate]  noexit :=
Req?req:admitted_req; <handling  req> ; stop
endproc (*Handler*)

endproc (*Handlers*)

process  Central_repository [int_gate]  noexit :=
...
...
endproc (*Central_repository *)

endspec (*ClientServer*)
```

9.4 Formal Description

9.4.1 Auxiliary Concepts

An occurrence of a process identifier P in a behaviour expression E is *guarded* in E if it occurs within some subexpression a;F of E, with $a \in Act \cup \{i\}$. Otherwise it is said to be *unguarded* in E [33].

A recursive process definition process P... := BExp endproc is said to be *guarded* if the process identifier P occurs guarded in BExp. Conversely, it is said to be *unguarded* if P occurs unguarded in BExp.

These definitions extend easily to the case of mutually recursive processes.

Note that the above definition of guarded recursion is weaker than the one given in [34], which excludes unobservable actions from valid guards. In this context we prefer the former, since some tools (e.g. the already mentioned interpreter described in [29]) can indeed cope with i-guarded recursion. Therefore, we will consider as acceptable a solution which transforms unguarded recursions in i-guarded recursions (Solution a below).

9.4.2 Input

PD: a process definition containing unguarded occurrences of process identifiers in the context of interleaving operators.

9.4.3 Output

QD: a guarded process definition

9.4.4 Transformation Requirements

The defining behaviour expression in QD should no more contain unguarded occurrences of process identifiers (while the rest of the process definition is to be left unchanged).

9.4.5 Correctness Preservation Requirements

PD and QD are observationally equivalent.

9.5 Solution

Solution a): an internal action is introduced in order to model the dynamic creation of new processes (this has been done, for example, in the already cited example of [19]). The internal action can be thought of as modelling the creation of a new process and can be explicitely used, in later development stages, as a placeholder for a call to an operating system "create_a_new_process" procedure.

The result is still a specification of the type considered as abstracting the need for an a-priori unbounded multiplicity of resources in Sect. 7.3.1, where a successive transformation is shown to bound the number of resources.

The solution therefore consists in substituting the unguarded occurrences of the process identifier P by i;P . In the case of mutually recursive processes it is enough to substitute in this way only some of the occurrences of the involved process identifiers, (only one of them in the case of strictly cyclical mutually recursive definitions).

$$\text{process P : noexit} := \; Q|||P \;\; \text{endproc}$$

$$\Downarrow$$

$$\text{process P : noexit} := \; Q|||i;P \;\; \text{endproc}$$

Note that this solution is based on the equivalence $X \approx i;X$, which is valid for observational equivalence. Since we perform this substitution inside an interleaving operator context (which preserves observational equivalence), the Correctness Preserving Relation of this solution is observational equivalence.

If the process identifier P occurs inside a choice context, the equivalence $P \approx i;P$ is valid only for weak trace equivalence: this means that this transformation can be applied also to more general input, with mixed interleaving/choice contexts for P, but loosing observational equivalence.

Solution b): no internal actions are introduced, but rather the existing actions are rearranged in order to eliminate the unguarded recursion. The solution applies the following transformation rule:

$$\text{process P : noexit} := \; []\{a_i;Q_i\}|||P \;\; \text{endproc}$$

$$\Downarrow$$

$$\text{process P : noexit} := \; []\{a_i;(Q_i|||P)\} \;\; \text{endproc}$$

Where by $[]\{a_i;Q_i\}$ we indicate the choice among the processes of the set.

This transformation rule preserves strong bisimulation equivalence and has the advantage to introduce no infinite sequences of internal events; on the other hand, this rule is more complex to be applied than the former solution, since it modifies at least two contexts enclosing the unguarded occurrence, as opposed to the simple addition af an i as a prefix for the unguarded occurrence.

9.6 Examples of solution

Solution a):
In the specification of the above example, an unobservable action is now introduced to explicit the process generation:

```
specification ClientServer[Req, Reply] : noexit :=

        (* Abstract Data Types Definitions*)

behaviour
hide int_gate in
(Handlers [Req, Reply, int_gate]
| [int_gate] |
Central_repository [int_gate])

where
process  Handlers [Req, Reply, int_gate]  noexit :=
i; Handlers [Req, Reply, int_gate]
|||
Handler [Req, Reply, int_gate]

where
process  Handler [Req, Reply, int_gate]  noexit :=
Req?req:admitted_req; <handling  req> ; stop
endproc (*Handler*)

endproc (*Handlers*)

process  Central_repository [int_gate]  noexit :=
...
...
endproc (*Central_repository *)

endspec (*ClientServer*)
```

As an alternative, we can extract from the Handler process its first action, namely the acceptance of a request, before the actual generation of the process (Solution b):

```
specification ClientServer[Req, Reply] : noexit :=

        (* Abstract Data Types Definitions*)

behaviour
hide int_gate in
(Handlers [Req, Reply, int_gate]
| [int_gate] |
Central_repository [int_gate])

where
process  Handlers [Req, Reply, int_gate]  noexit :=
Req?req:admitted_req;
(Handlers [Req, Reply, int_gate]
|||
Handler [Req, Reply, int_gate](req))

where
process  Handler [Req, Reply, int_gate]
(req:admitted_req) noexit :=
<handling  req> ; stop
endproc (*Handler*)

endproc (*Handlers*)

process  Central_repository [int_gate]  noexit :=
...
...
endproc (*Central_repository *)

endspec (*ClientServer*)
```