Correctness of the Promela models of MANIFOLD programs, I

Alain Fagot, Adriano Scutellà

CWI
Postbus 94079
1090 GB Amsterdam
`e-mail: {adriano, fagot}@cwi.nl`

1 Introduction

Coordination languages [Coo99] are used for the orchestration of various activities of a distributed, parallel application. A representative of this class of languages is **MANIFOLD** [Arb95,Arb96a,Arb96b]. Promela [Hol91] is a high level modelling language used for the specification of concurrent systems and it can be argued that it is also a sort of coordination language [Hol99]. Besides last observation the idea of this work comes from the fact that there is a correspondence between **MANIFOLD** constructs and Promela constructs, the use of Promela to specify a model of a **MANIFOLD** application may give insights in the comprehension of problems that have to be addressed when we want to verify such applications.

We point out that our interest is *the study of reachability of* **MANIFOLD** *program states*: By considering this problem we are led to take into account a broader class of related ones, such as deadlocks, livelocks, unreachable code.

A methodology in order to translate **MANIFOLD** constructs into Promela statements has been described in [FS99]. In that paper we address the problem of correctness of the choices described there.. In Section 4 of this paper we prove that the solutions we adopted are "correct" with respect to the behaviour of a **MANIFOLD** application. Section 2 contains two examples, some planning for future work and conlcusion is drawn in Section 5

2 MANIFOLD and its relation with Promela and Spin

This section is a sketchy outline of the characteristics of the coordination language **MANIFOLD**. The reader who wishes to learn more on **MANIFOLD** may consult [Arb95,Arb96a,Arb96b], for Promela and Spin the book written by Gerard Holzmann [Hol91] and the online documentation [WWWSP] are our references.

Like Promela applications, a **MANIFOLD** application consists of various running processes of some given types. We find two kinds of processes in **MANIFOLD**: *atomic* and *coordinator* processes. Atomic processes compute (i.e. accept data, compute and, possibly, produce some output data); a coordinator process has the responsability to coordinate the various activities that it has to supervise. Each process has a number of input and output *ports* which are its gateways to the external world. Processes use their ports to send or receive data. Data are transferred using *streams* connected to ports. A stream is a communication link that transports a sequence of bits, grouped into *units*.

It represents a reliable, unbounded and directed flow of information from its *source* to its *sink*. The constructor of a stream between two processes is, in general, a third (coordinator) process. Once a stream is established between a producer process and a consumer process, it operates autonomously and transfers the units from its source to its sink. There are four basic stream types; each type of stream behaves according to a slightly different protocol: See the MANIFOLD manual [Arb96b] for details. Because of its activity a stream is modelled in Promela by means of a process and of two channels (one for input, one for output).

Processes can communicate using an additional event mechanism: A process may *raise* an *event* which propagates to all processes belonging to its scope, the event will be saved in their *event memories*. These processes upon detection of the event generally react by perfoming some action.

Actions perfomed by atomic processes (which can be written in a host language) are production and consumption of data units through their ports, generation and reception of events and computation.

Coordination processes are written in MANIFOLD and they are called *manifolds*. Their body of consists of a finite number of *states*. Each state is identified by a *label* which is followed by the body of the state. The label of a state defines the condition under which a transition to that state is possible[1]. The body of a simple state defines the set of primitive actions that are to be performed upon transition to that state. The body of a compound state is either a nested block of states or a call to a (parameterized) subprogram known as a *manner* in MANIFOLD. Primitive actions perfomed by manifolds can be grouped into *communication actions* and *topology actions*. Actions in the first group consist of generation and reaction to events: Manifolds have ports but they have no instruction for manipulating units; ports are used as facilities over which to connect an incoming stream with an outgoing one in a pipeline fashion. Topology actions are those that change the structure of the application: Creation and activation of a process type instance, creation, connection and reconnection of streams to ports. At each instant a manifold is either executing *all* primitive actions specified in its state (and leave that state only when all actions have been perfomed) or trying to evaluate some condition specified in the labels in order to make a transition to another state.

A manifold process, coordinating the activity of some atomic processes, may itself be considered as an atomic process by another manifold process, giving rise to a sophisticated hierarchy of coordinators.

The execution of Promela instructions is limited to interleaving (of all possible instructions that can be concurrently perfomed by independent processes, only one is effectively executed while the others are delayed): This restriction, though peculiar for MANIFOLD, is not that limiting provided that we adapt MANIFOLD semantics to such restricted models. The execution of Promela programs is asynchronous, meaning, in the words of the language designer that: "No additional assumption is made on the relative speed of the process execution" and fits with MANIFOLD notion of asynchronicity.

3 Divide et impera

MANIFOLD encompasses two orthogonal characteristics. One is the event broadcasting and reaction mechanism which is the means to achieve coordination. A complementary

[1] It is an expression that can match observed event occurrences in the event memory of an instance.

characteristic is the possibility, beyond creation of processes, to dynamically rearrange the data flow network between various processes in the application. These two characteristics are quite different in nature and their treatment is also independent. In order to illustrate the approach used we believe that two examples are more illuminating than one, in addition to this, it is possible, to disentagle the proof of correctness of the models for both aspects of Manifold.

3.1 The broadcasting mechanism of Manifold

With the first application we give account of the broadcasting mechanism of Manifold and what is the Promela model that accomplishes it.

The Manifold code reported below consists of two processes that react to two events playing a sort of Ping-Pong game.

```
1  event ping, pong, wait.
2
3  manifold pinger
4  {
5    begin   : (raise(ping), post(wait)).
6    pong    : (raise(ping), post(wait)).
7    wait    : (preemptall, terminated(void)).
8  }
9
10 manifold ponger
11  {
12    begin   : (post(wait)).
13    ping    : (raise(pong), post(wait)).
14    wait    : (preemptall, terminated(void)).
15  }
16
17  process pi is pinger.
18  process po is ponger.
19
20 manifold Main
21 {
22  begin: activate(pi, po).
23 }
```

The application consists of three processes as depicted in Figure 1: The process `Main` activates (line `22`) the two instances of the process types `pinger` and `ponger` already instantiated at lines `17` and `18` and named `pi` and `po`, respectively. A process of type `pinger` has its body defined at lines `4 - 8`. As reaction to the predefined event `begin`, it enters the state labeled `begin` (line `5`), to execute the specified instructions . These are raising the event `ping`, to start the game, posting (i.e., raising only for itself) the event `wait`. Once all instructions are executed the state may be abandoned (preempted) and another search in the event memory is made. Upon detection of the event `pong` the process enters the state labeled `pong` and executes again the previous set of actions - see line `6`. The state labeled `wait`, line `7`, entered on detection in the event memory of the event `wait`, is a busy waiting cycle[2].

[2] This cycle is implemented with the two special primitives `preemptall` and `terminated(void)` which realize this busy wait: `preemptall` instructs the process

A process of type `ponger` (lines `10 ... 15`) behaves analogously to processes of type `pinger` except for the fact that as reaction to `begin` it only posts `wait`; while on reception of the event `ping` it raises a `pong`.

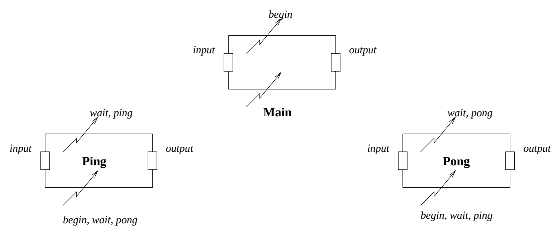

Fig. 1. Structure of the MANIFOLD application for Ping-Pong.

The intended semantics of MANIFOLD provides rules that establish which processes will receive an event: In principle an event is received by any process in the application, the process will react to the event if it does have a handle for it. In practice, only those processes which are interested in that event will receive it into their event memories for further treatment.

To model the event broadcasting each MANIFOLD process is expressed as a Promela process equipped with a *broadcast channel*, which will be used to receive all kinds of communication from other processes. Communication is exploited using messages of different types carrying some extra information. The type of information is partitioned into *control information* and *event information* (including predefined events). There are different *types* of messages, each type having a fixed fields structure: In those fields additional information such as the process identity is carried. The broadcast channel of a process, `p`, is known by all processes for which `p` is interested in receiving their events and by any other process that has to communicate with it (such a process is, for instance, the creator of `p`). A process in order to communicate with `p` will insert in the broadcast channel of `p` all messages directed to `p`. The model for Ping-Pong is pictured in Figure 2.

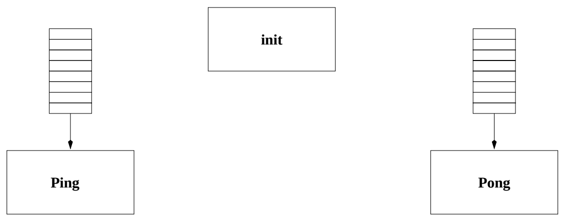

Fig. 2. Structure of the Promela application for Ping-Pong.

with the directive that any incoming event has to be reacted, `terminated(void)` realizes the waiting part, the process will wait for termination of the special process `void` (that never terminates) if no events are present in the event memory.

Messages used for broadcasting an event `ename`, have type prefixed with `ev_` followed by the name of the event, such as `ev_ename`. Each process collects messages from its broadcast channel (in a later stage it will treat them in the proper way). Every process, `p`, maintains also a list of *partners*: It consists of all process interested to receive the events raised by `p` (either because they expicitly signed-in, or because they are in the scope of `p`). In the Promela model the broadcasting of a raised event by `p` corresponds to sending to each partner in the list, trough their broadcast channel, of an event `ev_name`.

Each MANIFOLD application begins with a list of events and a list of process types that will be instantiated at run-time. To create the message types for all such events it suffices to scan them all and create a corresponding message. For the Ping-Pong application we have the following message types:

```
mtype   = {
          _csignin, _csignout, ev_begin       /*  predefined   */
            ev_ping, ev_pong, ev_wait         /* user  events  */
          };
```

Messages of type `_csignin`, `_csignout` are control messages corresponding to predefined constructs or events of MANIFOLD. The message `_csignin` is used to communicate to a process, `p`, that the source of the message, `s`, is interested in `p`'s events. The receiver `p`, will put the broadcast channel of `s` in the list of process partners to which it has to send its events.

Messages of type `_csignout`, serve the purpose to communicate to the receiver that the source of the message is not interested anymore in its events, the receiver will delete the process broadcast channel from its list of partners.

In MANIFOLD a process, `p`, is activated trough the execution of the statement `activate(p)`, this will correspond to the insertion of the special event `begin` in the event memory of `p` ; this event is always the first one to be reacted by an instance of a process. In the Promela model, to activate a process,`p`, a message of type `ev_begin` is put into its broadcast channel.

The drawings of Figure 3 show the internal structure of a MANIFOLD process and of its corresponding Promela model. A MANIFOLD process spends its life by repeatedly choosing a state to react. The choice is based on the labels of that state and on the event occurrences present in its event memory. If there are (many) event occurrences matching (many) labels, then one is chosen according to some strategy and the corresponding instructions specified in that state are perfomed. Instructions prescribed in the state have to be all executed before leaving the it.

The model in Promela behaves in an analogous way. Its behaviour is divided in two parts: The first part, tagged with a label `CONTROL`, is a loop. In this loop a process performs all those actions necessary in order to keep track of incoming events and to maintain the partners table. The event-memory is simulated by booloean variable flags that are set when an event occurrence is received. The `CONTROL` part is followed by the `CODE` part: It is also a loop. This part models the states that have to be entered in reaction to an event occurrence, it consists of a list of labels analogous to the list of labels encountered in the corresponding MANIFOLD code. Each time an event occurrence is reacted the loop is broken and the two phase cycle is repeated until the process terminates.

Each process type declaration in MANIFOLD (beginning with the keyword `manifold`) corresponds to a `proctype` declaration in Promela carrying some extra parameters

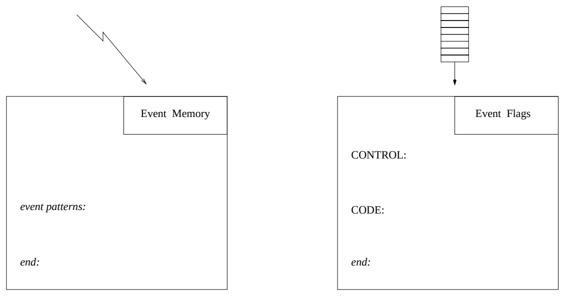

Fig. 3. Structure of MANIFOLD and Promela processes

needed to communicate the identity of the broadcast channel. Upon creation each process knows its broadcast channel which is passed to it by its creator in the formal parameter `ownbc`. For the process of type `pinger` of MANIFOLD the Promela declaration coming after the `mtype` declaration above is schematized as:

```
proctype pinger (chan ownbc)
{
 CONTROL:
         do ... od
 CODE:
         do ... od
 goto CONTROL

 end: ...
}
```

We find a name for the type, `pinger`, followed by the formal parameter for the broadcast channel `ownbc`. There can be other parameters corresponding to MANIFOLD parameters.

The `CONTROL` part for `pinger` is reported below:

```
CONTROL:
1      f=0;
2      do
3       :: ownbc?? _csignin(partnerbc) ->/* register signing-in processes */
...
10      ::ownbc??_csignout(partnerbc) -> /* forget signing-out processes */
...
15       /* check incoming events, set  event flags */
16      :: (empty(ownbc) && f == 0) -> goto end
17      :: ownbc?ev_ping(partnerbc)  -> atomic { ef_ping=true  ; f=f+1 }
18      :: /* similar code as the above line for other events */
19      :: (f > 0)  -> break     /* to break the loop */
20     od;
```

The process starts at the `CONTROL` loop checking whether it has incoming messages. The flag `f` is needed to eventually break the loop. Control messages are retrieved before others by means of the special selective receive with double question marks.

If the message is `_csignin(partnerbc)` the process registers the broadcast channel, `partnerbc`, of the signing-in process in a table `partners[]`.

If the message is `_csignout(partnerbc)` information about the process (the broadcast channel) emitting the `signout` is deleted from the table.

If the message is `ev_ename` then the corresponding flag `ef_ename` is set.

Remark 1. In case there are many events of the same type coming from different sources the event flags variables are structured into an array: one for each event and for each partner of the process so to maintain an event memory for all possible event occurrences that have been received. A more complete account on this can be found in the expanded version of this paper [FS99a].

Note that the `CONTROL` loop is broken either because the broadcast channel is empty or because the variable `f` is greater than 0. The variable `f` has also the purpose of counting the number of event occurrences received by the process. The condition "`empty(ownbc) && f == 0`" on line 18 says that a process can finish (`goto end`) its activity only if its broadcast channel is empty (no event or control messages) and it has no event occurrences to be treated. After the process exits the control part, it passes to the `CODE` part.

A Manifold process "chooses" one state on the basis of the label of that state and the received event occurrences. The `if .. fi` construct of Promela is used to model this choice. The construct presents a number of conditions enclosed between the keywords `if` and `fi` many of which may be true contemporarily, if more than one condition is true, one of them is chosen nondeterministically and the instructions specified are performed, this happens analogously in Manifold: If more than one enabling event is present in the memory, an enabled state is chosen nondeterministically. Conditions model the labels of a Manifold state. Since labels can match event occurrences conditions in the Promela model are specified by the boolean event flags variables set in the `CONTROL` part.

```
  CODE:
1       if
2       :: (ef_begin) -> /* begin event  received */
3              atomic { /*raise(ping)*/
4                      m=0 ;
5                      do
6                       :: (m < np) -> partners[m]!ev_ping(ownbc) ; m=m+1
7                       :: (m == np) -> break
8                      od ;
9                      ownbc!ev_wait(ownbc);
10                     ef_begin=false
11                     }
12
13      :: (ef_pong) ->  /* pong event  received */
14              atomic { /* same as ef_begin */ }
17
18      :: (ef_wait) ->  /* wait event  received */
19               {nempty(ownbc)-> ef_wait=false}
20      fi;
21      goto CONTROL ;
```

```
22 end: skip
```

The event flag variable `ef_begin` is used for the special predefined event `begin`; `ef_pong`, for the event `pong`; `ef_wait` for `wait`. For each of them there is a list of instructions corresponding to the MANIFOLD ones. They are enclosed in an `atomic` statement so that the set of instructions between the brackets of an atomic clause has to be executed as an indivisible piece of code (saving also time in validation of the model). The instruction `raise(ping)` is translated into a loop (lines `3` to `9`) in which a message of type `ev_ping`, containing the sender's identity and its broadcast channel, is sent to all processes registered in the table `partners[]` of the broadcaster: The table is scanned and a message is put into each broadcast channel retrieved from the table. In this case there is only one partner which is an instance of `pong`. Posting of an event (say `wait`) is modelled by sending the corresponding message type to the process own broadcast channel (corresponding instruction: `ownbc!ev_wait(ownbc)`).

The `init` process is analogous to the MANIFOLD process `Main`. Creation of a MANIFOLD process corresponds, in Promela, to the creation of the process and of its broadcast channel. Upon creation each process interested in the events of this new process is informed sending an appropriate `_csignin` type message that contains the identity and the broadcast channel of the process. Processes are activated sending the `ev_begin` message.

3.2 The model of a MANIFOLD stream

The behaviour of a MANIFOLD application, with the exception of the events control mechanism, is a game of dynamically creating new instances of processes and connecting or reconnecting their ports through streams. We consider now an application in which streams are set up, we will then analyze the Promela model; however, we introduce another style in writing a Promela model, namely, we use macro definitions (described in Table 1) which we developed with the aim of helping the user and to render the syntactic flavour of a MANIFOLD application. The MANIFOLD application computes Fibonacci numbers.

```
manifold PrintUnits() import.
manifold variable(port in) import.
manifold sum(event)
  port in x.
  port in y.
  import.
event overflow.

auto process v0 is variable(0).
auto process v1 is variable(1).
auto process print is PrintUnits.
auto process sigma is sum(overflow).

manifold Main()
{
 begin: (v0 -> sigma.x, v1 -> sigma.y, v1 -> v0,
                  sigma -> v1, sigma -> print).
19
20  overflow.sigma: halt.
21 }
```

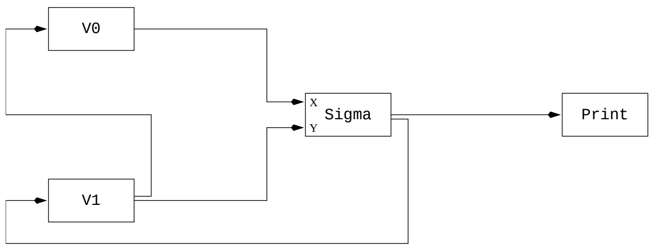

Fig. 4. Structure of the MANIFOLD application for computing Fibonacci numbers.

At run-time the structural view of the applicaton is as in Figure 4. The process named `sigma` is an atomic process declared at lines 3 trough 6. It computes the sum of two numbers supplied at its input ports `x` and `y`, until, upon reaching the overflow, it raises the event `overflow` and the application will terminate (line 20): The code of this process may not even be MANIFOLD code, it resides elsewhere as specified by the keyword `import`. Numbers produced by `sigma` are diverted to a process `print` that will print them. The two variable processes `v0` and `v1` will firtsly contain the two initial numbers of the Fibonacci sequence, 0 and 1, passed to them as parameters at their instantiation on lines 9 and 10; after they will serve as "containers" to feed `sigma` with the proper inputs it requires for the computation. The behaviour of the application is better explained in [Arb95] to which we refer the reader for further details. The Promela model with macros is listed below, the code is commented to achieve brevity in the presentation of the paper.

```
BEGIN_EVENTS            /* declaration of events */
EVENT(overflow)      /* corresponds to the declaration */
END_EVENTS         /* ``event overflow'' in the original */

#include "Streams.promela"       /* directives for the inclusion */
#include "printer.promela"          /* of  atomic and stream  */
#include "variable.promela"      /*      processes models        */
#include "sum.promela"


PROCESS(print,printer) /*instantiation of  process type printer name: print*/

PROCESS(V0,variable)   /*instantiation of process variable name: V0*/
PROCESS(V1,variable)   /*instantiation of process variable name: V1*/

PROCESS(sigma,sum)    /*instantiation of  process  type sum  name: sigma*/
PROCESS_PORT_IN(sigma,XPORT) /*input port declaration XPORT for sigma*/
PROCESS_PORT_IN(sigma,YPORT) /*input port declaration YPORT for sigma*/
```

```
BEGIN_MANIFOLD(main)          /*begin of the init (main) process*/

EVENT_MEMORY(overflow)
STREAM(0x)
STREAM(1y)
STREAM(s1)           /*local declarations needed for events and streams */
STREAM(10)
STREAM(sp)

INIT_PROCESS                  /*initialization of event memory, i.e.*/
INIT_EVENT(overflow)       /*initialization  event flags for each event*/

PORTS_MANAGEMENT                        /*management of  ports*/

EVENTS_MANAGEMENT                       /*management of the event memory*/
        EVENT_RECEIVING(overflow)

BEHAVIOUR
BEGIN_PATTERN_ESP(begin)    /* Manifold event pattern begin. */
    CONNECT_STREAM(KK,0x,V0,OUTPORT,Sigma,XPORT)  /*streams connections*/
    CONNECT_STREAM(KK,1y,V1,OUTPORT,Sigma,YPORT)
    CONNECT_STREAM(KK,10,V1,OUTPORT,V0,INPORT)
    CONNECT_STREAM(KK,s1,Sigma,OUTPORT,V1,INPORT)
    CONNECT_STREAM(KK,sp,Sigma,OUTPORT,Print,INPORT)
    ACTIVATE(Print,printer)                /*activation of processes*/
    ACTIVATE(V0,variable)
    ACTIVATE(V1,variable)
    ACTIVATE(Sigma,sum)
END_PATTERN_ESP(begin)              /*end of state labeled begin*/
END_MANIFOLD( main )                         /* end of main*/
```

Four our purposes, to model a stream it suffices to consider its behaviour abstracted away from the internal content of the stream. In the MANIFOLD reference manual [Arb96b] it is said that a stream is either *full* or *empty*. It is full if it contains at least one unit otherwise it is empty. There are four types of streams in MANIFOLD, these are BB, BK, KB and KK[3]. Another characteristic to be taken into account is the status of stream connection: The stream may be disconnected from both source and sink, connected to both of them, or connected to the source (or the sink) and disconnected from the sink (or the source). Thus a stream may be modelled trough a state based machine. In addition to that, the stream receives particular control messages from either its constructor (messages such as disconnect from source, disconnect from sink,

[3] The leftmost and rightmost letters refer to the behaviour of the two ends of the stream: BB, for instance, means *break-break*: Upon preemption (when the installator of the stream leaves the state in which a stream has been installed) the stream breaks automatically its connections. A KB type (*keep-break*) keeps the source connection and breaks the sink connection while a BK acts the other way round. A KK type never disconnects on preemption.

disconnect from both) or the source (sink) to which is attached for instance, to inform the stream about the death of the process or detach from some port.

Figure 5 reports the simplest state machine representation of a stream of type BB[4].

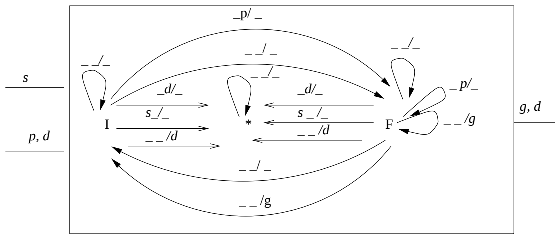

Fig. 5. State-machine of a BB stream.

The formalism used in Figure 5 has been introduced in [KSW97a,KSW97b]: It is briefly described in the appendix. The letter p indicates a *put* action (i.e., the transmission of a unit from a process into the stream). Analogously, g stands for *get*, it denotes the action of a process willing to get a unit from the stream. The letter d, denotes disconnection, s is the instalator preemption message. The stream can be in three different states: I: Empty, connected to the source, connected to the sink, F: Full, connected to the source, connected to the sink and *: The dead state, the stream is not active anymore.

Labels on arrows are *renamings* of the corresponding actions into boundaries observable actions: For instance, the transition $I \xrightarrow{p/\_} F$ means that upon a *put* action (executed by the source and no actions from the control and the sink) the stream will move from the empty state I to the full state F.

A stream of type BB is modeled by a process based on the state-machine representing the stream. This process manipulates its control and unit channels passed to it as parameters upon its creation. The reader may verify that the code defines a state based machine represented in Figure 5 (the code is included with the `#include ''Streams.promela''` directive).

```
 proctype streamBB ( chan i_channel, o_channel )
{
        int     values;
        Unit    unit;
        values = 0;
I:
        do
        :: i_channel?_sput(unit) -> /* put unit */
                o_channel!_sunit(i_channel, unit) ;
                values++ ; goto F
        :: i_channel??_spreempt -> goto end  /* preempt  */
        :: i_channel??_sdissource -> goto end /* disconnect */
```

[4] We omit for the sake of brevity the description of all types of streams which appear in the full version of this paper.

```
        :: o_channel??_sdissink -> goto end   /* disconnect */
        od;
F:
        do
        :: i_channel?_sput(unit) -> /* put unit */
                o_channel!_sunit(i_channel, unit) ;
                values++
        :: i_channel?_sget -> values-- ; /* get unit */
                if
                :: (values < 1) -> goto I
                fi
        :: i_channel??_spreempt -> goto end   /* preempt  */
        :: i_channel??_sdissource -> goto end  /* disconnect */
        :: o_channel??_sdissink -> goto end   /* disconnect */
        od;
end:   skip    /* handle disconnection from source and sink */
}
```

To connect multiple streams to one port, we use two different strategies depending on the sort of port and problem to be solved. For an input port, we need to get a unit from *any* stream connected to it, while for an output port we need to put a unit on *every* stream connected to it.

An input port is modeled as a Promela channel in which every stream connected to that port will put units. This input-port channel is passed as parameter on the instantiation of the Promela process modelling the stream. It is also passed to the process modelling the manifold as a field of the control message `_cconnect`

An output port is modeled as a list of Promela channels to which the process will have to send the message `_sput` to put a unit in the connected streams. These channels are passed as arguments of the control message `_cconnect`.

The code corresponding to what we have described is reported in Table 1 as the following macro definitions: `PUT(INPORT)`, `GET(OUTPORT)`.

4 Adequacy of the Promela models

In this section we try to address the problem of the adequacy of the Promela models we designed. The idea is to show that the behaviour of the Promela Model M_P and the behaviour of the MANIFOLD application are "the same".

As pointed out in the preceding sections MANIFOLD presents two orthogonal characteristics: The event-broadcasting and the set up of streams. For reasons of space we cannot handle the correctness of both aspects, we have choosen to treat the most delicate one which is the modelling and behaviour of streams. The broadcasting mechanism of MANIFOLD will be treated elsewhere: Here we mention that a different strategy, in order to implement broadcasting, would have been to use some global variables. Each time an event (occurrence) is raised the corresponding global variable b is set to true, each process interested in those event could look at b and react, but the process should reset the variable for all other interested processes. The reader understands that not even taking into account the overhead necessary to implement what we said, this approach could cause some inconsistency; for instance, a problem could be to determine

Manifold pseudo-manifold	Promela
`raise(event)` RAISE(event)	m=1; do ::(m==np) -> break ::else -> partners[m]!EV_(event)(ownbc); m++ od;
`post(event)` POST(event)	ownbc!EV_(event)(ownbc);
STREAM(name)	chan SCC_(name) = [QSZ_CC] of {mtype};
`process name is manif.` PROCESS(name , manif)	chan BC_(name) = [QSZ_BC] of {mtype, chan, int}; chan SUC_PORT(name,INPORT) = [QSZ_UC] of {mtype, chan};
`activate(name)` ACTIVATE(name , manif)	BC_(name)!_cconnect(SUC_PORT(name,INPORT),INPORT); BC_(name)!!_csignin(ownbc); BC_(name)!ev_begin(ownbc); m=0; do ::(m==np) -> np++; partners[m]=BC_(name); break ::else -> if ::(partners[m]==BC_(name)) -> break ::else -> m++ fi od; run manif (BC_(name));
PUT(port)	m=0; do ::(m==PNS_(port)) -> break ::else -> PCC_(port)[m]!!_sput; m++ od;
GET(port)	PUC_(port)?_sunit(partner); partner!_sget;
`fproc.fport->tproc.tport` CONNECT_STREAM(type , name , fproc , fport , tproc , tport)	BC_(fproc)!_cconnect(SCC_(name), fport); BC_(tproc)!_cconnect(SUC_PORT(tproc,tport),tport); run STREAM_(type)(SCC_(name),SUC_PORT(tproc,tport));
`(preemptall, terminated(void)).` WAIT	if ::(gnf>1) -> skip ::else -> nempty(ownbc) fi;

Table 1. Translation of MANIFOLD constructs into Promela instructions.

which process is the last one to react to an event; in this case b is not supposed to be set anymore. Another solution could be to have, for each process p, a separate process m_p implementing its memory. In this case for each **MANIFOLD** process we would have two Promela processes. The solution we adopted it has been suggested from the study of the **MANIFOLD** compiler. A proof of its adequacy can be carried out in the same style of the one we present now for streams.

Correctness of the stream model. A stream, S, is created and set by a coordinator, C, between the output and input ports of a source process, O, and a sink process, I, respectively. Once connected the stream operates autonomously taking the units appearing at the output port of O and delivering them (in First In First Out order) to the input port of I when it executes a read operation on the port. In addition to the put-unit and get-unit operations the stream is sensible to other signals namely the disconnection of ports, and the preemption messages from C that instruct the stream to disconnect from either source or sink or both (depending on the stream type). We consider here the stream type `BB`, as represented in Figure 5. The system we consider are represented in the uniform framework offered by **Span(RGph)** in Figure 6

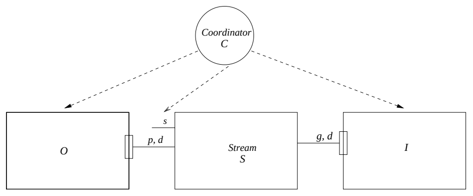

Fig. 6. Model of a stream connection in **MANIFOLD** and Promela.

Both models can be described at first by the same equation:

$$\mathbf{M} = O \bullet S \bullet I \tag{1}$$

In order to distinguish between the two models we may use the two subscripts M and P to denote which model we are referring to. We ignore the coordinator C from that equation, in fact it simply creates the stream and sends the preemption signals to all three components. That is the only way in which a coordinator can affect the behaviour of a connection, since we are interested in the behaviour of the system in steady-state we need not to consider the singular effects of C. Denote with $\mathbf{M}_P$ and $\mathbf{M}_M$ the two systems. We show that there exists a 2-cell $\alpha\colon \mathbf{M}_P \to \mathbf{M}_S$ which is an abstraction and preserves behaviours (as advocated in [AL91]), i.e. we want to prove:

Theorem 1. *Given the two sytems $\mathbf{M}_P$ and $\mathbf{M}_M$ there exists an abstraction $\alpha\colon \mathbf{M}_P \to \mathbf{M}_S$ such that $\mathcal{B}(\mathbf{M}_P) \subseteq \mathcal{B}(\mathbf{M}_M)$.*

Proving the theorem amounts to construct the abstraction α which we do in the reamining part of this paper. In Section 3 we describe a model of a stream as a system comprising a process and two channels, if we indicate with IC, S and OC respectively the input channel, the stream process and the ouput channel, the following equation describes a stream:

$$S_P = IC \bullet S \bullet OC \tag{2}$$

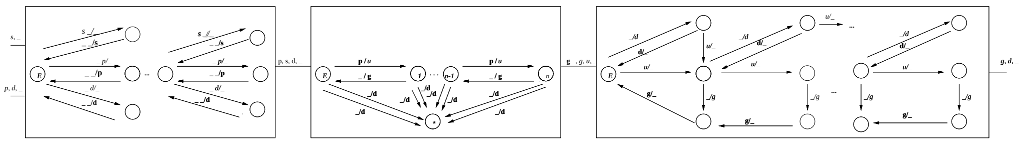

Fig. 7. Model of a stream connection in Promela.

The evaluation of the expression in Equation 2 is the system in Figure 7.

The reader may have noticed that the graphs for IC and OC are not completely drawn, howewer we give a presentation for those graphs which are finite anyway.

A vertex of IC, or OC, represents the set of messages currently stored into the channel and is denoted below with a set theoretic representation; $\overline{s} = \{m_1, m_2, \cdots, m_k\}$ is the point at which the set of messages $\{m_1, m_2, \cdots, m_k\}$ has been received. The graph for IC is described by the following equations

$$\begin{array}{lll}
E \xrightarrow{s\_/\_} \{s\}, & \overline{s} \xrightarrow{s\_/\_} \overline{s} \cup \{s\}, & \overline{s} \cup \{s\} \xrightarrow{\_\_/\mathbf{s}} \overline{s} \\
E \xrightarrow{\_p/\_} \{p\}, & \overline{s} \xrightarrow{\_p/\_} \overline{s} \cup \{p\}, & \overline{s} \cup \{p\} \xrightarrow{\_\_/\mathbf{p}} \overline{s} \\
E \xrightarrow{\_d/\_} \{d\}, & \overline{s} \xrightarrow{\_d/\_} \overline{s} \cup \{d\}, & \overline{s} \cup \{p\} \xrightarrow{\_\_/\mathbf{d}} \overline{s}
\end{array}$$

The graph for OC is presented by:

$$\begin{array}{lll}
E \xrightarrow{u/\_} \{u\}, & \overline{s} \xrightarrow{u/\_} \overline{s} \cup \{u\}, & \overline{s} \cup \{s\} \xrightarrow{\_\_/\mathbf{s}} \overline{s} \\
E \xrightarrow{\_/d} \{d\}, & \overline{s} \xrightarrow{d} \overline{s} \cup \{d\}, & \overline{s} \cup \{p\} \xrightarrow{\_\_/\mathbf{p}} \overline{s} \\
 & \overline{s} \cup \{u\} \xrightarrow{\_/} \overline{s} \setminus \{u\} \cup \{g\}, & \overline{s} \cup \{g\} \xrightarrow{\_\_/\mathbf{g}} \overline{s}
\end{array}$$

Above, the letter s stands for the preemption message from the coordinator, u is the exchange of a unit, p denotes a put and g a get operation, d is the disconnection. Boldface letters ($\mathbf{g}$, $\mathbf{p}$, $\mathbf{g}$) denote the read counterpart actions. Once we have constructed the abstraction α we only need to show that indeed is a 2-cell in **Span(RGph)**. It is convenient to define the abstraction α for each component and rely on the structure of **Span(RGph)**.

The abstraction α has three main components: $\alpha =< \alpha_O, \alpha_S, \alpha_I >$ each of them may be still decomposed if needed. Figure 8 depicts the abstraction function. The first thing worth noticing is that with respect to the source and sink processes O and I we need to consider those transitions that affect the behaviour of the stream, we put: $\alpha_O = id_O$ and $\alpha_I = id_I$, that is the abstraction is the identity on O and I. Each transition of the Promela components O and I is mirrored on the MANIFOLD component. In the picture we depicted only those transitions which affect for the stream behaviour, that is the p (put) and d (disconnect) operations executed by OC-components and the g (get) and analogous d operations executed by IC components. What remains to be defined is that part of α that acts on the stream. Since a model of a stream is made of three components we define the 2-cell on each of them separately and rely on the structure of **Span(RGph)**. Figure 9 depicts the action of the 2-cell on states and transitions of the stream components of α. The notation is taken from [LS] dotted arrows depict the transition mappings, while dashed arrows depict the state mapping, of course an explicit analitic definition is possible but we find more suggestive this notation.

By exhibiting a 2-cell we completed the proof of the theorem.

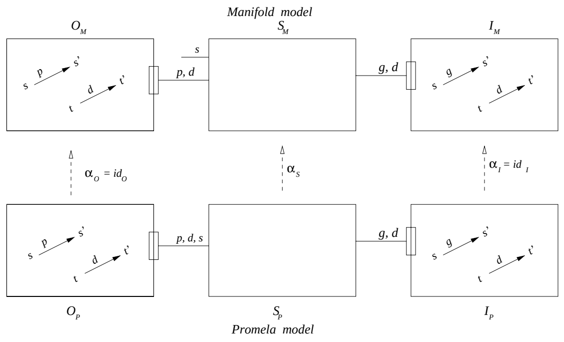

Fig. 8. The abstraction α: $\mathbf{M}_P \to \mathbf{M}_M$.

5 Final remarks

In this paper we addressed the problem of the correctness of the models presented already in [FS99], in part II of this paper we treat the event broadcasting mechanism whcih for lack of space we could not expose here. The proof of correctness is based on abstraction morphisms on the graph-theoretic representations of the models, these morphisms preserve behaviours. The outcome of this study is the adequacy of the models we devised, it can then be used to adapt the MANIFOLD compiler in such a way that it can also produce (based on the methodology presented in [FS99]) the relevant Promela code for the MANIFOLD application. Another useful line of further study is the tracking back to the original application of the output traces produced by Spin. Once we have proven the correctness of our approach, we are implementing the Promela model of a large case study presented in coordination [CNT98] which has been solved in [Scu99] using MANIFOLD.

Thinking is common to everybody.
(Eraclitus, In: Strobeus, *Florilegius* 3, 1, 179.)

Acknowledgements. Special thanks go to Kees Everaars, Freek Burger and Kees Blom for answering all questions we have been asking them regarding the MANIFOLD compiler.

References

[AL91] M. Abadi, L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2), pp.: 253-284, May 1991.

[Arb95] F. Arbab. Coordination of massively concurrent activities. Technical Report CS–R9565, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, November 1995. Available on-line http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z.

[Arb96a] F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer-Verlag, April 1996.

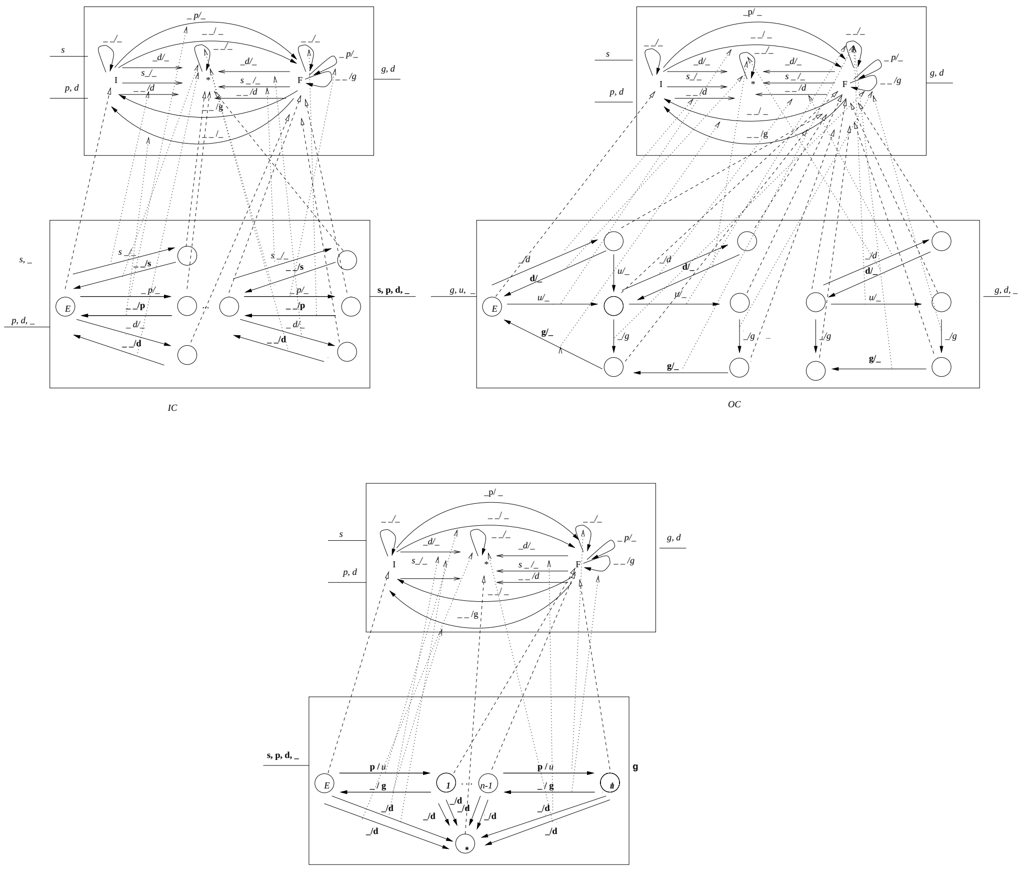

Fig. 9. The 2-cell $\alpha_S\colon S_P \to S_M$.

[Arb96b] F. Arbab. MANIFOLD version 2: Language reference manual. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1996. Available on-line http://www.cwi.nl/ftp/manifold/refman.ps.Z.

[Ben67] J. Bènabou, Introduction to bicategories, Reports of the Midwest Category Seminar, Lecture Notes in Mathematics 47, pages 1–77, Springer-Verlag, 1967.

[CNT98] P. Ciancarini, O. Niestratz, and R. Tolksdorf. A case study in coordination: Conference Management trough the Internet. Electronic note.

[Coo99] P. Ciancarini, A.L.'Wolf, editors. *3rd Int. Conf.on Coordination Languages and Models*, volume 1594 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1999.

[FS99] A.Fagot, A.Scutellà. Model-checking of MANIFOLD applications with the Spin model-checker. In: *Prooceedings of the 5th Workshop on Theoretical Aspects of Spin*, Trento, Italy, 1999.

[FS99a] A.Fagot, A.Scutellà. Model-checking of **Manifold** applications with the Spin model-checker (extended version). In preparation.

[Hoa] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice Hall International.

[Hol91] G. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, Englewood Cliffs, pp.: xii-500.

[Hol99] G. Holzmann, Private communication, 1999.

[KSW97a] P. Katis, N. Sabadini, R.F.C. Walters, Span(Graph) a Categorical Algebra of Transition Systems, LNCS 1349, pp.: 307-321, 1997.

[KSW97b] P. Katis, N. Sabadini, R.F.C. Walters, Representing Place/Transitions nets in Span(Graph), LNCS 1349, pp.: 322-336, 1997.

[LS] F.W. Lawvere, P.H.Schanuel *Conceptual Mathematics*

[Scu99] . A. Scutellà newblock *Simulation of a conference Management System using* **Manifold**. LNCS 1594, pp.: 243-258, 1999.

[WWWSP] AA.VV Spin on line documentation . Available on line at `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

A The bicategory Span(RGph)

The base category we use is the one of reflexive graphs **RGph**.

Definition 1. *The bicategory* **Span(RGph)** *is defined as follows:*

- *Objects are directed reflexive graphs.*
- *An arrow from X to Y consists of a graph H together with a pair of graph morphisms $X \xleftarrow{l_H} H \xrightarrow{r_H} Y$.*
- *Composition of spans* $\mathbf{H} = X \xleftarrow{l_R} R \xrightarrow{r_R} Y$ *and* $\mathbf{S} = Y \xleftarrow{l_S} S \xrightarrow{r_S} Z$ *is the span;* $\mathbf{H} \bullet \mathbf{K} = Y \xleftarrow{l_H p_1} R \bullet S \xrightarrow{r_S p_2} Z$ *formed taking the pullback* $R \xleftarrow{p_1} S \bullet S \xrightarrow{p_2} S$*, it is called a span from X to Z.*
- *The identity arrow for an object X is the span* $\mathbf{X} = X \xleftarrow{1} X \xrightarrow{1} X$*, 1 being the identity arrow on X.*
- *A 2-cell from $X \xleftarrow{d} H \xrightarrow{e} Y$ to $X \xleftarrow{f} H \xrightarrow{g} Y$ is a graph morphism $\alpha \colon H \to K$ such that $f\alpha = d$ and $g\alpha = e$. Vertical composition $\circ$ of 2-cells is composition of graph morphism. Horizontal composition is the unique arrow generated by the pullback property.*
- *Vertical and horizontal composition are related by the middle-four interchange law:* $(R \bullet S) \circ (T \bullet Z) = (R \circ T) \bullet (S \circ Z)$.

Given the span $\mathbf{H} = X \xleftarrow{l} H \xrightarrow{r} Y$ we call H the *head* of the span; X and Y the left and right *feet* of the span, respectively; the two morphism l, r are the left and right *legs* of the span.

Objects can be also given as products of graphs. Figure 10 shows some representation for spans. On the left there is the span $\mathbf{H} = X_1X_2 \longleftarrow H \longrightarrow Y_1Y_2Y_3$, the drawing on the right depicts the composition $\mathbf{H} \bullet \mathbf{K}$ with $\mathbf{K} = Y_1Y_2Y_3 \longleftarrow K \longrightarrow Z$.

The identity span is denoted by a plain wire.

The composition above defines the composite of spans $R \colon X \longrightarrow Y$ and $S \colon Y \longrightarrow Z$ as a graph $R \bullet S$, the head of the span whose vertex set is:

$$\{< r, s >| \ r \text{ is a vertex of } R,\ s \text{ is a vertex of } S \text{ such that } r_R(r) = l_S(s)\}$$

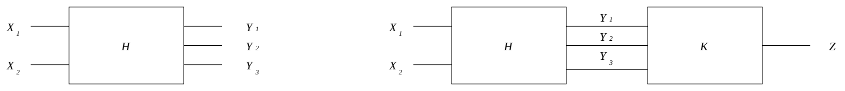

Fig. 10. Another way to depict spans

and whose edge set is:

$$\{<\rho,\sigma>|\ \rho \text{ is an edge of } R,\ \sigma \text{ is an edge of } S \text{ such that } r_R(\rho)=l_S(\sigma)\}$$

Definition 2. *The tensor of a pair of objects X and Y is their product in* **Span(RGph)** *that is: $X \otimes Y = X \times Y$.*

The tensor of two spans $\mathbf{H} = W \xleftarrow{l_H} H \xrightarrow{r_H} X$ and $\mathbf{K} = Y \xleftarrow{l_K} K \xrightarrow{r_K} Z$ is the span $\mathbf{H} \otimes \mathbf{K} = \langle H \otimes K, l, r\rangle = W \times Y \xleftarrow{l_H \times l_K} H \otimes K \xrightarrow{r_H \times r_K} X \times Z$.

The tensor can be depicted as in the following picture:

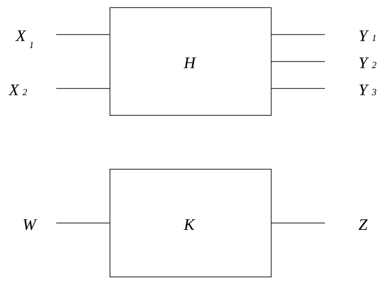

Fig. 11. Geometric view of the tensor of two spans

Constants of the algebra. Beyond the above operations, there are also some constants are useful in order to describe general systems.

There is a span with head X and legs 1_X:$X \to X$, Δ_X: $X \to X \times X$ called the diagonal of X (denoted as Δ_X: $X \to X \times X$). The codiagonal is the span Δ^*_X: $X \times X \to X$).

Projections are spans with head $X \times Y$ and legs $1_{X\times Y}$:$X \times Y \to X \times Y$ and p_X: $X \times Y \to X$, herep_X is the projection arrow, it is depicted by the termination of the wire Y. There is a similar reverse arrow denoted p^*_X.

Given a pair of objects A and B, there is a graph morphism tw: $A \times B \to B \times A$ that acts as follows: $\langle a,b\rangle \mapsto \langle b,a\rangle$. The *permutation* $\pi_{A,B}$: $A \otimes B \to B \otimes A$ is defined to be the span $A \times B \xleftarrow{1_{A\times B}} A \times B \xrightarrow{tw} B \times A$.

The terminal graph, denoted by I, is the graph with one vertex and one edge (by necessity the identity loop). The unit of the self-dual compact-closed structure on **Span(RGph)** is η_X: $I \to X \times X$, it is the span with head X and legs $!$: $X \to I$, Δ:$X \to X \times X$. The co-unit is ε_X: $X \times X \to I$.

Ususally, when it will be clear in the context, we will omit subcripts in the above simbology.

The geometric representation of these constants is given in Figure 12.

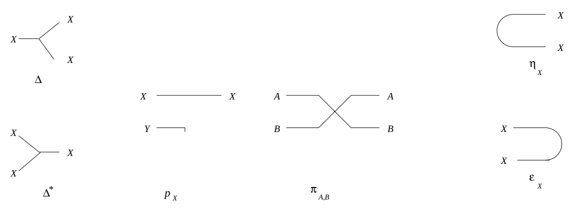

Fig. 12. Constants of the algebra

Definition 3. *A valid behaviour for* $\mathbf{R} = R\colon X \to Y$ *is a path* π *in* R *beginning at its initial vertex and ending at some final one. The set of behaviours is denoted by* $\mathcal{B}(\mathbf{R})$.

Applying the legs l_R and r_R of the span to a valid behaviour π yields a pair of behaviours: one $l_H(\pi)$ in X and another $r_H(\pi)$ in Y. These induced behaviours may be thought as the behaviours of the boundaries of the system R. We have the following result.

Theorem 2. *Consider spans* $\mathbf{R} = R\colon X \to Y$, $\mathbf{S} = R\colon Y \to Z$ *and* $\mathbf{T} = R\colon X' \to Y'$.

- *A valid behaviour of* $\mathbf{R} \bullet \mathbf{S}$ *is a pair of valid behaviours* $< \rho, \sigma >$, ρ *being a valid behaviour of* $\mathbf{R}$ *and* σ *one of* $\mathbf{S}$ *which agree on the common boundary, that is:* $r_R(\rho) = l_S(\sigma)$.
- *A valid behaviour of* $\mathbf{R} \otimes \mathbf{T}$, *is a pair of valid behaviours* $< \rho, \tau >$, ρ *being a valid behaviour of* $\mathbf{R}$ *and* τ *one of* $\mathbf{T}$.
- *A valid behaviour of* $\eta_X\colon I \to X \times X$ *is a path* π *in* X *reflected equally on the two boundaries. (Analogously for* ε*.)*
- *A valid behaviour of* $\Delta_X\colon X \to X \times X$ *is a path* π *in* X *reflected equally on the three boundaries. (Analogously for* Δ^**.)*