On the Specification of Shared Objects in Π

J. Cramer, H. Schumann

October 17, 1991

Abstract

Critical points in the current version of the Π-Language [CS90] are the treatment of shared objects and the object configuration specification. Both features of the Π-Language have lead to major difficulties in the past. Concerning shared objects we found out that the semantics of the share-statement is quite unclear and as a consequence also their specification. Concerning the object configuration specification it has turned out that this view as a whole is incomprehensible and hard to motivate.

In this paper we propose an alternative approach in which the sharing of objects is specified on the type level by introducing 'shared' types. Only objects of these explicitly marked types can be used as shared objects. How objects are then actually shared is specified in the imperative view specification of single CEMs.

The introduction of shared types has two consequences. The first is that the separate object configuration specification is no longer necessary. This reduces the syntax of the configuration specification by more than two thirds. The second concerns the treatment of shared objects in distributed modular systems. To handle object configurations in such systems it is necessary to introduce 'virtual' CEMs on top of the configuration specification. These are CEMs which are responsible for the installation of shared objects in such configurations, but need not necessarily to be implemented.

Contents

1 Introduction

In the current version of the Π-Language [CS90] we describe the sharing of objects by means of so-called configuration actions in the object configuration specification. In contrast to usual CEM operations, configuration actions describe system management activities, like eg.

- creation of initial object configuration,
- sharing of objects,
- manipulation of a given object configuration.

These configuration actions should be performed only at specific points in time (eg. system initialization, system update) by only some authorized persons (eg. system manager, superuser). Due to the large similarities in the description of CEM operations and configuration actions - in both cases we use imperative algorithms - the object configuration specification is often hard to motivate and should be replaced by a more intuitive description scheme.

Another drawback of the current version is that on the type level there is no information available about the sharing of objects. As a consequence it is not possible to determine correctness and compatibility properties concerning shared objects based on type specifications (type view specifications and type configurations). Such tests are only possible if the corresponding object configuration is specified.

In the following we discuss an alternative approach in which the sharing of objects is specified on the type level by introducing 'shared' types. A positive side-effect of this approach is that the object configuration specification is no longer necessary.

2 Basic Idea

The basic idea of the new approach is to distinguish on the type level between 'normal' and 'shared' types. The difference between both is that only objects of the latter can be used as shared objects. All other objects can only be used exclusively.

The distinction between 'normal' and 'shared' types makes it necessary to instantiate data type specifications, as 'normal' types in one case and as 'shared' types in the other case. For example if we have a CEM which needs a local and a shared buffer, we can use the same buffer specification in two instantiations. Thus we have not only CEM incarnations due to different actualizations of a CEM but also data type incarnations due to its usage as 'shared' or 'normal' data type.

In the following we describe first how shared objects/types are used and described within a single CEM and then how configurations are built containing shared objects/types.

To illustrate this approach we use a slightly modified version of the well-known producer-consumer example, in which we have two subsystems communicating via a shared object. Both subsystems have a similar structure. They consist of a local part, a local buffer and a shared buffer (see also figure 1).

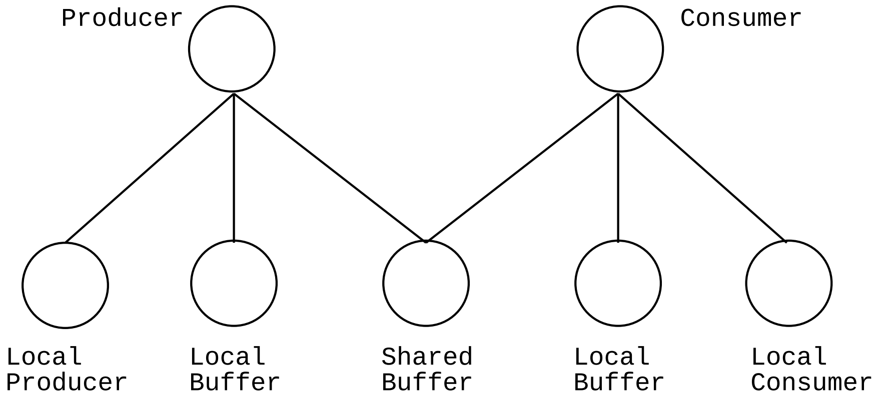

Figure 1: Object Configuration of the Producer-Consumer Example

During the following chapters we sketch the specification of this example but concentrate ourselves on the shared object/shared type aspect.

3 Shared Objects/Types in Single CEMs

3.1 Type View Specification

The only thing we have to do in the type view specification of a single CEM is to distinguish between sorts which are shared and sorts which are not shared. Due to the idea that each data type specifcation can be instantiated either as 'shared' sort or as 'normal' sort, this distinction is only specified in the import section of a CEM and not in the export interface. Because the common parameters are part of the export interface we therefore restrict the import of shared sorts to the import section of a CEM.

On the syntactical level this results in the following modification of rule T6 [CS90]:

T6: <tv_cem_import> ::=

["**shared**"] "**sort**" <sort_name>
["**general description**" <comment>]
{ <operation_description> }*
["**local operations**" <local_operation_list>]

If we consider the type view specification of the CEM `PRODUCER` in our producer-consumer example we obtain the following specification skeleton:

```
cem PRODUCER

type view specification

export

  sort Producer

    operation initialize_producer: Local_prod, Local_buffer, Shared_buffer -> Producer

    operation start_produce: Producer -> Producer

    ...

body

  sort Producer

  construction of Producer is Triple
    where
      Item1 <- Local_prod,
      Item2 <- Local_buffer,
      Item3 <- Shared_buffer
    end where

  operation initialize_producer: Local_prod, Local_buffer, Shared_buffer -> Producer
    variables
      lp: Local_prod;
      lb: Local_buffer;
      sb: Shared_buffer
    equations
      initialize_producer (lp, lb, sb) = make_triple (lp, lb, sb)

  operation start_produce: Producer -> Producer

  ...

import

  sort Local_prod

    operation create_local_prod: -> Local_prod

    ...
```

```
  sort Local_buffer

    operation create_local_buffer: -> Local_buffer

    operation read_local_buffer: Local_buffer -> Data

    operation write_local_buffer: Data, Local_buffer -> Local_buffer

    ...

  shared sort Shared_buffer

    operation create_shared_buffer: -> Shared_buffer

    operation read_shared_buffer: Shared_buffer -> Data

    operation write_shared_buffer: Data, Shared_buffer -> Shared_buffer

    ...

  sort Triple

    operation make_triple: Item1, Item2, Item3 -> Triple

    operation get_first: Triple -> Item1

    ...

  sort Data

    ...

end cem PRODUCER
```

According to this import section specification only objects of sort Shared_buffer can be used as shared objects within the body specification of the CEM `PRODUCER`.

3.2 Imperative View Specification

Due to the strict conformance between type and imperative view we distinguish in the imperative view between shared types and non shared types. As a consequence we have similar changes in the syntax description as in the type view:

I6: <iv_cem_import> ::=

["**shared**"] "**type**" <type_name>
["**general description**" <comment>]
{ <procedure_description> }*

The respective specification of the CEM `PRODUCER` looks as follows:

```
cem PRODUCER

imperative view specification

export

  type Producer

    procedure initialize_producer (in lp: Local_prod;
                                      lb: Local_buffer;
                                      sb: Shared_buffer)
                                   returns Producer

    procedure start_produce (inout p: Producer)

    ...


body

  type Producer

    construction of Producer is Triple
      where
        Item1 <- Local_prod,
...
        Item2 <- Local_buffer,
...
        Item3 <- Shared_buffer,
...
      end where

    procedure initialize_producer (in lp: Local_prod;
                                      lb: Local_buffer;
                                      sb: Shared_buffer)
                                   returns Producer
      begin
        return make_triple (lp, lb, sb)
      end

    procedure start_produce (inout p: Producer)

    ...

import

  type Local_prod

    procedure create_local_prod returns Local_prod

    ...

  type Local_buffer
```

```
    procedure create_local_buffer returns Local_buffer

    procedure read_local_buffer (in lb: Local_buffer) returns Data

    procedure write_local_buffer (in d: Data inout lb: Local_buffer)

    ...

  shared type Shared_buffer

    procedure create_shared_buffer returns Shared_buffer

    procedure read_shared_buffer (in sb: Shared_buffer) returns Data

    procedure write_shared_buffer (in d: Data inout sb: Shared_buffer)

    ...

  type Triple

    procedure make_triple (in i1: Item1; i2: Item2; i3: Item3) returns Triple

    procedure get_first (in t: Triple) returns Item1

    ...

  type Data

    ...

end cem PRODUCER
```

Up to now we have introduced those changes which are due to the distinction between shared and normal types. In the next step we have to define how the actual sharing of a specific object is described. This is done in the procedure bodies, where we deal with control flow, an abstract notion of storage and the creation and deletion of objects.

Objects are created at the beginning of a procedure execution according to the declare-statement and are deleted at the end of the procedure execution in which they are declared. During their lifetime they can be used in the construction of more complex objects, as parameters in other procedure invocations, or simply as stores for intermediate results.

The important difference between shared and normal objects is their way to become part of a more complex object. In case of normal objects only a copy of the object becomes part of the more complex object. This has the consequence that a normal object can never be a part of different more complex objects. In contrast to this a shared object is directly used to build a more complex object. This allows that one (shared) object is at the same time part of different more complex objects, and that a shared object can survive (as part of another object) the end of the procedure execution in which it is declared.

From a more technical view the difference between both kind of objects is the parameter passing mechanism in case of in-parameters. Normal objects are passed through by a call-by-value semantics. This means that each time a normal object is used as in-parameter the invoked procedure works conceptually with a copy of this object. This ensures the strict encapsulation principle, on which the Π-Language is based, because we have only well-defined[1] side-effects in the resulting systems.

In contrast to this shared objects are passed through using the call-by-reference semantics. This mechanism allows that one and the same object can be part of different complex objects at the same time.

To illustrate this let us consider the procedure `create_prod_cons` in our producer-consumer example, which creates the initial object configuration of our producer-consumer system (see figure 1).

```
body

  ...

  procedure create_prod_cons (in lp: Local_prod;
                                 lc: Local_cons)
                              returns Prod_cons_tuple
    declare
      lb  : Local_buffer;
      prod: Producer;
      cons: Consumer
    shared objects
      sb  : Shared_buffer
    end declare

    begin
      prod := initialize_producer (lp, lb, sb);
      cons := initialize_consumer (lc, lb, sb);
      return make_tuple (prod, cons)
    end

  ...
```

The object which is returned at the end of the procedure has the structure depicted in figure 2.

As we can see this structure has one shared object, the object `sb`, which is explicitly declared as shared object in the delare-statement at the beginning of the procedure `create_prod_cons`. The local buffer object `lb`, which is not declared as shared object, appears as two different objects in the resulting object configuration. This is because the constructing procedures `initialize_producer` and `initialize_consumer` use copies of the object `lb` but references in case of the shared object `sb`.

On the syntactical level we need only a simple modification of rule IP8:

[1] Well-defined means here that side-effects are limited to the one inout-parameter of an object modifying procedure.

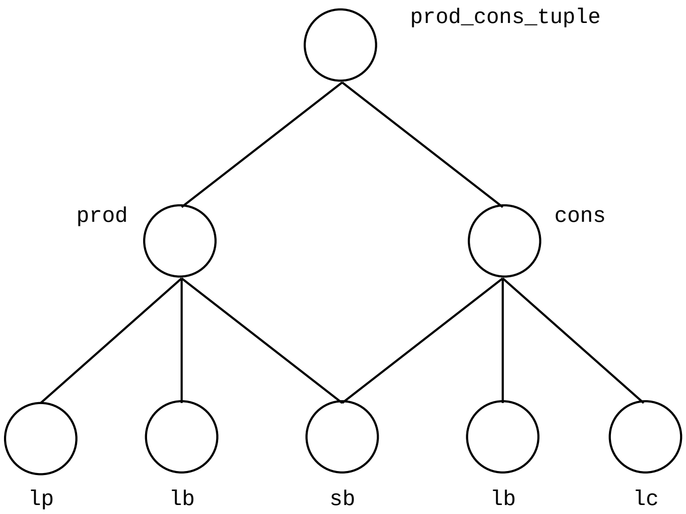

Figure 2: Return-Object of the Procedure create_prod_cons

IP8: <object_declaration> ::=

> "**declare**"
> <object_declaration> { ";" <object_declaration> }*
> "**shared objects**"
> <object_declaration> { ";" <object_declaration> }*
> "**end declare**"

This explicit distinction of shared and non shared objects in the declare-statement is to some degree redundant, because the type information implies whether an object is a shared one or not. But we prefer here an explicit specification because the sharing of objects is an error-proning issue which makes a careful and disciplined use necessary.

4 Shared Types in Configuration Specifications

The aforementioned changes are the only ones needed on the level of single CEMs. But with this little syntactical modifications we have reached the goal that information about the sharing of objects is available on the type view level and we need no longer an object configuration specification. Thus the configuration specification consists only of the type configuration.

The necessary modifications in the type configuration specification depend mainly on the fact that we have to build component incarnations not only due to different actualizations of a component but also due to the possible instantiation of a data type specification as shared or not-shared sort. Only a sort which is instantiated as a shared sort (see example below) can be used to actualize a 'shared sort' import requirement.

Together with the omission of the object configuration we have got major changes in the configuration specification, which results in a complete redesign of the syntax description. The complete syntax for the configuration specification is given in Appendix A. At this place we sketch only the configuration specification of our producer-consumer example. The interesting point here is the instantiation of the CEM `BUFFER` as a shared buffer and as a non shared buffer in the component incarnation statement. Only the component incarnation `shared_buffer` with the explicit marked sort `Buffer` as shared sort can be used to actualize the import requirements of the producer and consumer specifications, where we had explicitly specified the need for a shared sort `Buffer`.

The classification of component incarnations as `virtual` and `root components` in the following example is optional and explained in chapter 5.

```
configuration PRODUCER_CONSUMER_SYSTEM

  component incarnations

      prod_cons     : Prod_cons  is virtual;

      producer      : Producer   is root;
      consumer      : Consumer   is root;

      shared_buffer : Buffer     with shared sort Buffer;

      local_buffer  : Buffer;
      producer      : Producer;
      consumer      : Consumer;
      local_prod    : Local_prod;
      local_cons    : Local_cons;
      triple        : Triple;
      tuple         : Tuple;
      string        : String

  component connection

    connection of prod_cons

    ...

    connection of producer

      from local_prod import

        sort Local_prod <- Local_prod

        operations
          create_local_prod <- create_lp
          ...

      from local_buffer import

        sort Local_buffer <- Buffer
```

```
    operations
      create_local_buffer <- create_buffer;
      read_local_buffer   <- read_buffer;
      write_local_buffer  <- write_buffer;
      ...

  from shared_buffer import

    sort Shared_buffer <- Buffer

    operations
      create_shared_buffer <- create_buffer;
      read_shared_buffer   <- read_buffer;
      write_shared_buffer  <- write_buffer;
      ...

  from triple import

    sort Triple <- Triple

     operations
      make_triple <- make_triple
      get_first   <- get_first
      ...

  from string import

    sort Data <- String

  ...

end configuration PRODUCER_CONSUMER_SYSTEM
```

5 Virtual Top CEMs for Distributed Modular Systems

If we investigate the described approach more closely we see that we need always an upper level CEM which declares and introduces the shared objects for lower level object configurations. As far as we consider only not distributed modular systems this makes no problems. But in case of distributed systems we have to answer the question how to specify a shared object for the distributed parts of a system.

Our solution for this problem is the introduction of virtual CEMs on top of distributed modular systems. These virtual CEMs model an entity which has to some degree an overview and overall control of the complete distributed system. These CEMs have not to be implemented as usual programs but reflect more the work of a system manager who has control on global names or can install communication facilities etc.[2].

[2]Nevertheless in some cases it can be possible that parts of such a virtual CEM are implemented

If we come back to our producer-consumer example we can imagine that the producer should be located on one site and the consumer on another site. Thus the top CEM `PROD_CONS` does only exist as a virtual part of our system (see figure 3).

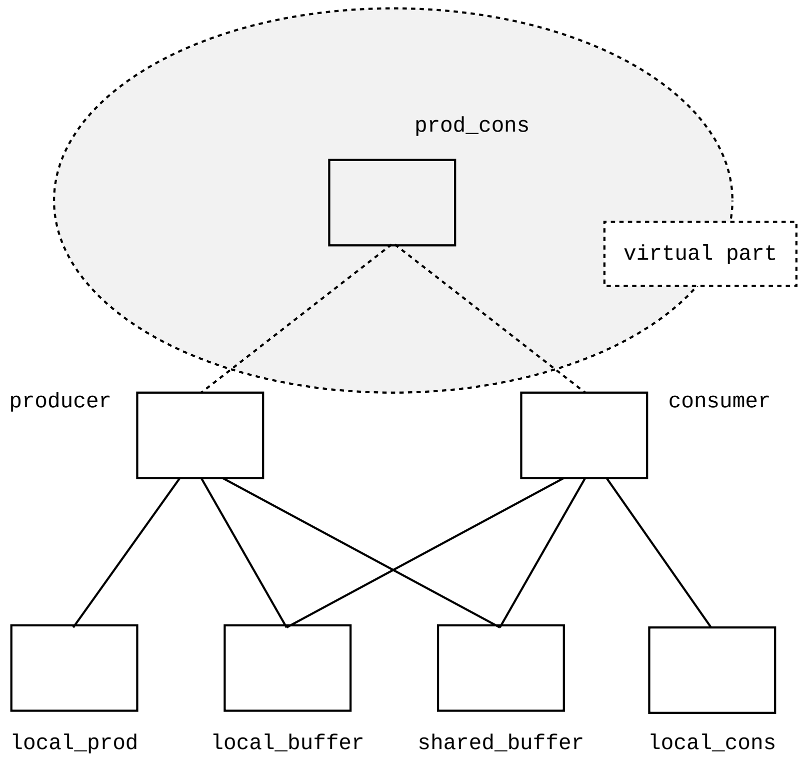

Figure 3: Virtual Part of a Distributed Modular System

To specify that a CEM is only a virtual one we modify the syntax rule G2 as follows:

G2: <cem_specification> ::=

”**cem**” <cem_name>
[”**is virtual**”]
[”**general description**” <comment>]
[<type_view_specification>]
[<imperative_view_specification>]
[<concurrency_view_specification>]
”**end cem**” <cem_name>

on one or different sites of a distributed system. But usually at least some part will be realized by hardware (global name services) or via global system management.

According to this rule we have the following specification of the CEM PROD_CONS:

```
cem PROD_CONS

  is virtual

  general description { ... }

  type view specification

    ...

  imperative view specification

    ...

  concurrency specification

    ...

end cem PROD_CONS
```

In the configuration specification we can identify virtual CEMs by an optional attribute `is virtual` in the component incarnation part (see example in chapter 4).

6 Summary and Open Questions

In this paper we propose a new approach to handle the specification of shared objects in Π. This approach is based on the idea to make the information of object sharing available on type level and on the level of single CEMs.

On the syntactical level there are only minor modifications necessary which do not complicate the syntax of the language. On the contrary due to the omission of the object configuration specification the complete syntax of the Π-Language is reduced by approximately 35 rules. This is more than two thirds of the complete configuration specification of the old version 2.0.

From the point of use we have got the impression that the handling of shared objects is much easier if we can specify as early as possible which objects are shared and which not. In this new approach this is possible as well on type level as on the object level within single CEMs.

Furthermore this approach allows the dynamic sharing of objects. In contrast to the old version, where the sharing of objects is specified explicitly by a share-action in a configuration action, objects can now be shared dynamically at arbitrary points during the execution of a program (if they are declared as shared objects). This broadens the field of applications, but is on the other side a potential new source of errors.

This approach gives also rise to some open questions. These are centered around the notion of a virtual CEM on the one hand side and on the other side around the notion of a shared sort on type level.

Concerning virtual CEMs we identify the following open questions:

- How many virtual CEMs may exist in a modular system? Do we allow virtual CEMs in subconfigurations or do we have at most one virtual CEM in a complete system?
- What is the semantics of virtual CEMs and how do we realize them?

Questions around the notion of shared sorts concern especially the formal algebraic basis but address also pragmatic problems:

- How can we use the sharing information on type level to determine correctness and compability properties?
- What is the meaning of an import specification? Is it the specification of a data type as seen from the importing CEM (relative requirement) or is it an absolut requirement for the complete system? Using an example, does a stack specification in the import section require a non shared stack incarnation (due to the equations `top(push(s,i)) = i` and `pop(push(s,i)) = s`) or can we use also a shared stack incarnation to fulfill these stack requirements?

Another open problem is related to the disconnect-statement of the old version. There this statement should describe how two object configurations which are linked by a shared object could be separated (disconnected). The question is whether we need an equivalent statement in this new approach and how it could look like.

Altogether we can say that this new approach seems to be a real improvement compared to the old version although we have up to now made no experiences in larger systems.

A Syntax Description of the Configuration Specification

General Description:

G2: <configuration_specification> ::=

> "**configuration**" <configuration_name>
> ["**general description**" <comment>]
> <type_configuration>
> "**end configuration**" <configuration_name>

Type Configuration

ST1: <type_configuration> ::=

> <component_incarnation_specification>
> [<component_connection_specification>]

ST2: <component_incarnation_specification> ::=

> "**component incarnations**"
> <component_id_declaration> { ";" <component_id_declaration> }*

ST3: <component_id_declaration> ::=

> <component_id_list> ":" <component_name>
> [<incarnation_attribute>]

ST4: <incarnation_attribute> ::=

> "**is virtual**" |
> "**is root**" |
> "**with shared sort**" <sort_name_list>

ST5: <component_connection_specification> ::=

> "**component connection**"
> { <component_connection_description> }$^+$

St6: <component_connection_description> ::=

”**connection of**” <component_id>
{ <actual_import> }$^+$

ST7: <actual_import> ::=

”**from**” <component_id> ”**import**”
{ <actualization> }$^+$

ST8: <actualization> ::=

[”**import of**” <component_id>]
{ <data_type_actualization> }$^+$

ST9: <data_type_actualization> ::=

<sort_actualization>
<operation_actualization_list>

ST10: <sort_actualization> ::=

”**sort**”
<sort_name> ”←” <sort_name>

ST11: <operation_actualization_list> ::=

”**operations**”
<operation_actualization> { ”;” <operation_actualization> }*

ST12: <operation_actualization> ::=
<operation_name> ”←” <operation_name>

References

[CS90] J. Cramer and H. Schumann. Syntax Description of the Π-Language with Examples (Version 2.0). Technical Report No. 49, University of Dortmund, Dept. of Computer Science, Software-Technology, July 1990.