Implementation of “Generic Synchronization Policies” in Pict

Patrick Varone
Software Composition Group

IAM-96-005
April 1996

Abstract. This report presents an implementation of McHale’s “Generic Synchronization Policies” (GSP) in the Pict programming language. Since Pict is defined by mapping programming language constructs to the pi calculus, this exercise helps us on the one hand to assign an operational semantics to GSP, and on the other hand to explore the use of Pict and the pi calculus as a basis for modelling advanced object-oriented programming constructs.

Keywords: Object-based concurrency, synchronization, π calculus, semantics.

CR Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent programming; D.1.5 [Programming Techniques]: Object-Oriented Programming; D.2.m [Software Engineering]: Reusable Software; D.3.1 [Programming Languages] Semantics; D.3.2 [Programming Languages] Concurrent, distributed and parallel languages.

Author’s address: Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. E-mail: `scg@iam.unibe.ch`.
WWW: `http://iamwww.unibe.ch/~scg`.

1 Introduction

This report presents an implementation of the “Generic Synchronization Policies” (abbreviated as GSP) introduced in [4] using the language Pict. The main goal of this work was to see how well suited Pict is for implementing higher level abstractions. The remainder of this report is structured as follows: Section 2 briefly introduces the GSP concept. Pict [6] [7] [8] and a possible object model are presented in section 3. The implementation of GSP is the heart of section 4. Finally, Section 5 mentions future possible work.

1.1 Typesetting conventions

Throughout this report we use different fonts and styles to express code. Programs in Helvetica are expressed in a Pascal-like language while programs in `Courier` represents real Pict code. The Pascal-like language is used to show examples of GSP as well as the algorithms to implement it. Programs that can be run by the “extended Pict” compiler will always be written in `Courier`. By “extended Pict” we mean the Pict compiler plus the preprocessor presented in the

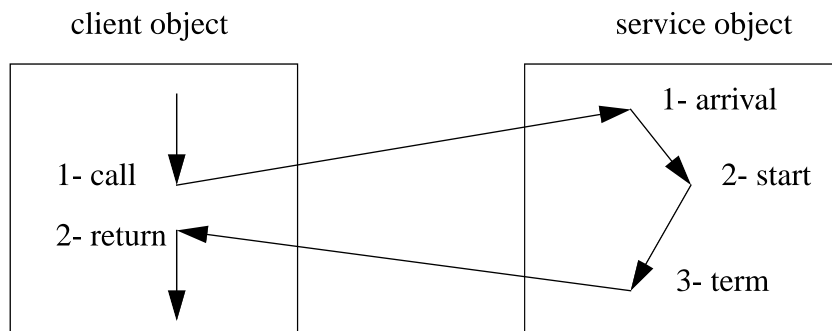
section 4. In the syntax presentation we use `Italic Courier` to denote a syntactic category and `normal Courier` for keywords, separators and operators.

2 Generic Synchronization Policies

2.1 Definition

This section briefly describes the GSP concept. We will not present the underlying "Service-object Synchronization" paradigm (Sos) in detail. For more information on Sos please refer to [4]. In what follows we assume that communication between objects is carried out by Remote Procedure Calls (RPC), that is a process invoking an operation of an object will be suspended until the operation is finished (and the possible return value accessible). However, it is important to note that this assumption does not belong to the Sos paradigm itself but has only been made to simplify the discussion. GSPs are used to synchronize the different method invocations of a particular object. In order to show how this mechanism works we first need to describe the sequence of events that takes place when one object invokes an operation upon another object.

From the service object's perspective (which is the only one we will consider), there are three events of interest: *arrival*, *start* and *term* (short for *termination*). When an invocation *arrives* it may be delayed due to synchronization constraints. Some time later it will *start* execution; and finally it will *terminate* execution. Here we assume that events do not overlap. For example if two invocations *arrive* at the same time then we assume that their *arrival* events will be ordered. The sequence of events is summarized in the following picture:



GSPs permit an action (user code) to be associated with each possible event. The execution of an action will always complete before another event can occur. Synchronization constraints between methods are expressed using the concept of a *guard* (i.e a boolean expression). An invocation will be delayed until the corresponding guard evaluates to true.

In GSP genericity lies in the fact that actions and guards are not associated directly with a particular method of a given object. Conceptually, methods are grouped into categories for which actions and guards are specified. At instantiation time the methods associated with one category will "inherit" its actions and guards.

In order to express complex synchronizations, we need access to information about the invocations upon the object. In code for actions and guards the following information will be made available:

- The arrival time of the *current* invocation (for which the action or guard is executed)

- The number of waiting invocations from a given category
- The number of executing invocations from a given category
- A list of all waiting invocations from a given category
- The method's parameters

(We could add other information like, for example, the number of terminated invocations or the list of all executing invocations from a given category. In our implementation we restricted ourselves to the five points mentioned above although it could be trivially extended to include this information as well).

2.2 Examples

The first example is the well-known *Reader-Writer* policy where we have two categories of methods: one category representing methods that only read instance variables of the object and another category representing methods that change the value of some instance variables. With GSP, the Reader-Writer policy could be written as follows (the syntax used is the one from [4]).

```
policy ReadersWriters[ReadOps,WriteOps]

function ReaderAllowed(t:Invocation):Bool
begin
    return exec(WriteOps) = 0;
end

function WriterAllowed(t:Invocation):Bool
begin
    return exec(ReadOps)+exec(WriteOps) = 0;
end

map guard(ReadOps) -> ReaderAllowed
    guard(WriteOps) -> WriterAllowed

end policy
```

This policy specifies that a *Read* operation can only take place when there is no executing *Write* operation (i.e. exec(WriteOps) = 0) and a *Write* operation can only take place when no other operation is executing (exec(ReadOps)+exec(WriteOps) = 0). The construct

```
map guard(ReadOps) -> ReaderAllowed
    guard(WriteOps) -> WriterAllowed
```

binds the guards for the different method categories. A guard (or an action) accepts one parameter of type Invocation that is bound to the current invocation for which the guard is evaluated. This Invocation object is useful to implement a FCFS (first-come, first-served) policy, for example

```
policy FCFS[AnyOps]

function OpAllowed(t:Invocation):Bool
begin
    for p in waitingList(AnyOps) do
        if p.arr_time < t.arr_time then
            return false
        endif;
```

```
        end;
        return true;
end
map guard(AnyOps) -> OpAllowed
end policy
```

The expression waitingList(method-category) returns the list of waiting invocations for the category method-category. To retrieve the arrival time of an invocation we can use the dot notation and the field name arr_time.

We have seen in the first example that we had access to the number of waiting or executing invocations with the two predefined exec(ReadOps) and exec(WriteOps) counters. Actually we do not really need them to be predefined as the following example shows

```
policy ReadersWriters[ReadOps,WriteOps]
var   execReadOps : Int := 0;
      execWriteOps : Int := 0;
procedure incReadOps(t:Invocation)
begin
      execReadOps := execReadOps+1
end
procedure decReadOps(t:Invocation)
begin
      execReadOps := execReadOps-1
end
procedure incWriteOps(t:Invocation)
begin
      execWriteOps := execWriteOps+1
end
procedure decWriteOps(t:Invocation)
begin
      execWriteOps := execWriteOps-1
end
function ReaderAllowed(t:Invocation):Bool
begin
      return execWriteOps = 0;
end
function WriterAllowed(t:Invocation):Bool
begin
      return execReadOps+execWriteOps = 0;
end
map   start(ReadOps) -> incReadOps
      term(ReadOps) -> decReadOps
      guard(ReadOps) -> ReaderAllowed
      start(WriteOps) -> incWriteOps
      term(WriteOps) -> decWriteOps
      guard(WriteOps) -> WriterAllowed
end policy
```

Our two predefined counters have been replaced by two variables that are updated when an operation starts and terminates execution, i.e. at the events *start* and *term*. The corresponding

actions are incReadOps and decReadOps for the category ReadOps, incWriteOps and decWrite-Ops for the category WriteOps. Of course, the same mechanism can be used to implement the waiting(WriteOps) counter, but this time the counter will be updated at the events *arrival* and *start*. Note that actions and guards are executed in mutual exclusion which allows a safe update of the variables execWriteOps and execReadOps.

3 Pict

Pict is a language based on the polyadic π-calculus [5] where the basic entities are *processes* and *channels*. The following paragraphs constitute an informal presentation of the language that should help the reader to understand section 4. More information can be found in [6].

3.1 Channels and types

A channel is a port over which one process may communicate with another. Every channel must be created before it can be used. The expression

```
new x : ^Int
```

creates a fresh new channel and binds it to the name `x`. The type of this channel is `^Int` where `^` is a channel type constructor. `Int` specifies that only integer values can be transmitted over this channel. There exist two other channel type constructors: `!` and `?`. `!` is used for an output-only channel and `?` for an input-only channel. Besides channels Pict defines the following primitive types: integers, booleans, characters, strings, records and tuples.

A tuple is written in square brackets with values separated by commas:

```
[1,"coucou",true]
```

The type of this tuple is `[Int,String,Bool]`. To select a value from a tuple we use pattern-matching. For example the expression

```
val [a,b,c] = [1,"coucou",true]
```

binds `a` to `1`, `b` to `"coucou"` and `c` to `true`.

A record is a collection of named field separated with commas:

```
record x=2,y='a' end
```

The type of this record is `Record x:Int,y:Char end`. To select a field from a record we use the dot notation.The expression

```
val z = (record x=2,y='a' end).y
```

binds `z` to `'a'`.

A type can be given a name by using a type declaration of the form

```
type name = type-definition
```

We can also define type operators (like `^`) using the following syntax

```
type type-operator = Fun(type-parameter) type-definition
```

as in the expressions

```
type Pair = Fun(X) [X,X]
val x : Pair Int = [2,3]
```

3.2 Processes

A process can be either a *sender* process or a *receiver* process.

The syntax of a sender process is

```
channel-name!value
```

whereas a receiver process is

```
channel-name?pattern > process
```

or

```
channel-name?*pattern > process
```

A *sender* process can only communicate with another *receiver* process that is ready to accept a value along the same channel. For the communication to take place the two processes must be put in parallel, as in

```
x!3 | (x?y > printi!y)
```

Here we have one sender process `x!3` and one receiver process `x?y > printi!y` put in parallel (the | operator). When the communication takes place, the pattern `y` will be bound to the value `3` and the sender process will be discarded. The receiver process will evolve into the process `printi!3` (where `y` has been replaced by `3`) so that the whole expression reduces to

```
printi!3
```

This sender process will in turn communicate with the Pict environment (where the channel `printi` is defined) which results in the display of 3. Actually the complete Pict program is

```
new x
run x!3 | (x?y > printi!y)
```

because we have to create any channel that we use in the program. Note that it is not necessary to specify the type of the channel `x` because it will be inferred by the Pict type system. The `run` directive allows a process expression to be put in parallel with the rest of the program, so that our example could also have been written as

```
new x
run x!3
run (x?y > printi!y)
```

As we have seen in 3.1, tuples of values can be used to transmit more than one value at a time

```
new x
run x![3,4] | (x?[y,z] > printi!y | printi!z)
```

In this case the values `3` and `4` will be displayed (but in an unspecified order). It is important to note that the parallel composition of two processes (the | operator) is itself a process.

Consider now the following program:

```
new x
run x!3 | x!4 | (x?y > printi!y)
```

The result of the program will be the display of either `3` or `4`, but not both because the process `x?y > printi!y` can only be used (communicate) once. So the whole expression reduces either to

```
x!3 | printi!4
```

or

```
x!4 | printi!3
```

If we want both values to be printed, we can use two copies of the same process but actually there is a better solution that consists in using so-called *replicated input* processes:

```
new x
run x!3 | x!4 | (x?*y > printi!y)
```

The replicated input process `x?*y > printi!y` behaves exactly as before except that once it has communicated on the channel `x` it will immediately create a new copy of itself. The expression

```
x!3 | x!4 | (x?*y > printi!y)
```

will therefore evolve into (assuming that `x!3` has been chosen for the first communication)

```
x!4 | printi!3 | (x?*y > printi!y)
```

and then into

```
printi!3 | printi!4 | (x?*y > printi!y)
```

Here the process `x?*y > printi!y` acts as a *server* process that repeatedly accepts a number on the `x` channel and prints it.

3.3 Local declarations

So far every channel in the program had a global scope. To create nested scopes we can use the `let ... in ... end` construct. The following example shows how a channel can be made local to a process:

```
run let new x in x!3 end
| let new x in x?y > printi!y end
```

This program prints nothing because the name `x` refers to a different channel in each process. It is however possible to export a channel outside of its scope of declaration:

```
new z
run let new x in z!x | x!3 end
| let new x in z?x > x?y > printi!y end
```

Here we have used a global channel `z` to transmit the channel `x` from the first process to the second one. This program will therefore print `3` as expected. Note that the local channel `x` in the second process is not used at all.

3.4 Derived forms

The previous definitions form what is usually called *core* Pict. We could stop defining new language constructs here and start programming with this tiny language. However, it appears that for most programming tasks the syntax presented so far is at too low a level to be used conven-

iently. For this reason, Pict defines a number of higher-level syntactic constructs that are then translated into core Pict.

3.4.1 The def construct

Assume we want to define a set of processes that read a number from a channel (different for each process) and then print it. All these processes behave in the same way. The only thing that changes is the name of the input channel. A convenient way to define such a family of processes is to use the `def` construct:

```
def myprint[c] > c?n > printi!n
new x,y,z
run x!1 | y!2 | z!3 | myprint![x] | myprint![y] | myprint![z]
```

With this definitions `myprint![x]` is equivalent to `x?n > printi!n` and `myprint![y]` is equivalent to `y?n > printi!n`. Internally, the expression

```
def name pattern > process
```

translates to

```
new name
run name?*pattern > process
```

so that the above definition of `myprint` becomes

```
new myprint
run myprint?*[c] > c?n > printi!n
```

3.4.2 Functions as processes

In Pict a function is implemented by a process that expects input values plus a reply channel to which it can send the computed result. Assume we want to define a function `plusone` that increments the value of an integer by one. We can write this function as follows:

```
def plusone[n,r] > r!(n+1)
new r
run plusone![3,r] | (r?x > printi!x)
```

Here the channel `r` plays the role of the reply channel upon which the result can be read. This program just "calls" the function `plusone` and, in parallel, waits for the result on `r` and prints it. This kind of function definition is so common that Pict provides for a special notation:

```
def function-name [x1,...,xn] = value
```

which translates to

```
def function-name [x1,...,xn,r] > r!value
```

The "type" of this function is in fact the type of the channel *`function-name`* which is

```
![X1,...,Xn,!R]
```

where X_i represents the type of x_i and `R` the type of the return value. The first `!` means that a function is an *output-only* channel (although it is used as an input channel in the function definition). This makes sense because we don't want to allow another process to input a value on *`function-name`*, which would amount to having two definitions for the same function.

Pict also defines a convenient notation to retrieve the value of the function. Whenever a value is expected (e.g. an integer), a function call of the form

```
function-name [x1,...,xn]
```

can be substituted for the value, provided `function-name` has been defined like this

```
def function-name [x1,...,xn] = value
```

and `value` is of the right type. Note that we drop the `!` character in the function call. Using this notation, we can rewrite our example as follows

```
def plusone[n] = n+1
run printi!(plusone[3])
```

Moreover, it is possible to define "anonymous" functions that are used just once

```
run printi!((abs [n] = n+1 end)[3])
```

Here the form

```
abs abstraction end
```

translates to

```
let def x abstraction in x end
```

3.4.3 Infix operators

New infix operators can be introduced in the language. They are defined exactly like functions but with their name enclosed in parentheses:

```
def (%%)[n,m] = (m+n)*2
run printi!(5%%4)
```

An infix operator can also be called using the usual syntax

```
def (%%)[n,m] = (m+n)*2
run printi!((%%)[5,4])
```

3.4.4 Sequencing

Assume we want to print the number `3` followed by the number `4` in that order. We cannot use a program like:

```
run printi!3 | printi!4
```

because we cannot tell which process will communicate first. Pict provides a simple way to call functions in sequence. The only requirement put on the function is that its result must be an empty tuple. For example, the *sequential* counterpart to `printi` is `prInt` which can be used as follows (`skip` is a process that does nothing)

```
run prInt[3];prInt[4];skip
```

Here `;` serves as a "statement" separator. The general pattern for a sequence is

```
function-name [x1,...,xn];process
```

which translates to

```
let
    val [] = function-name [x1,...,xn]
in
    process
end
```

It is important to note that the type of a *sequential* function will always be of the form

```
![X1,...,Xn,![]]
```

where `[]` is the type of the empty tuple `[]`. The type `![]` can be abbreviated as `Sig`.

3.5 The object model

Currently an object in Pict can simply be modelled as a record . For example, a mutable cell with two methods `set` and `get` can be implemented like this

```
def ref[init] = let
        new current
        run current!init
    in
        record
            set = abs [v,c] > current?_ > current!v | c![] end,
            get = abs [r]> current?x > current!x | r!x end
        end
    end
val r = ref[0]
run prInt[r.get[]];r.set[5];prInt[r.get[]];skip
```

The process `current!init` acts as the private instance variable holding the cell’s value and the two fields `set` and `get` contain methods to access and set this value. It is important to note that this very simple object model takes neither inheritance nor self-references in method definitions into account.

4 Implementation of GSP in Pict

This section describes the implementation of GSP in Pict, which is a more or less straightforward adaptation of McHale’s own implementation. Although Pict is not really object-oriented, it provides for a simple object model based on records. Actually one of the main aspects of the GSP concept (and its underlying paradigm) is that it is not tied to a particular language and can be fairly easily adapted to any object model. This section is divided into four subsections. The first part describes the enhanced object model while the three other subsections concentrate more on the implementation of GSP.

4.1 Synchronization wrappers in Pict

4.1.1 Synchronization wrappers and GSP

As mentioned above, the GSP mechanism is not tied to any object model. In fact the object model merely depends on the general scheme used to implement synchronization constraints among methods. Here we have opted, as in McHale’s thesis, for the commonly used *synchronization wrappers*. The main idea of the synchronization wrappers is to take a pure unsynchro-

nized object and to wrap its methods in a pre- and post-synchronization code. For example a method like:

```
method m1()
begin
      body;
end
```

will be transformed (wrapped) into

```
method m1()
begin
      "pre-synchronization code";
      body;
      "post-synchronization code";
end
```

Let's assume an RPC mechanism. Then the pre-synchronization code is responsible for executing the actions associated with the *arrival* and *start* events described in §2.1. This code is also responsible for suspending the calling thread until the guard associated with the method becomes true. The post-synchronization, for his part, executes the action associated with the *term* event.

Now consider the ReadersWriters policy described in §2.2 and suppose m1 belongs to the ReadOps category. Then m1 will be wrapped as follows:

```
method m1()
      var state:InvocationType;
begin
      state := pre_synch_ReadOps();
      body;
      post_synch_ReadOps(state);
end
```

The variable state represents information about the current invocation, e.g the arrival time mentioned in §2.1. pre_synch_ReadOps and post_synch_ReadOps stand for the synchronization code associated with the ReadOps category. Among other things, this means that those two functions will be used for every method belonging to the ReadOps category.

4.1.2 Binding a GSP to an object

Before explaining how an object can be bound to a particular policy, it is necessary to introduce a more convenient notation for objects in Pict. (This is the first extension of the Pict syntax. The translation to Pict is done via a special-purpose preprocessor.) The syntax of an object definition becomes

```
obj
    {var var-name : type = initial-value}
    {method method-name [[x1,...,xn]] = body end}
endobj
```

where {text} represents zero or more occurrences of text. This translates to

```
let
    {val var-name = ref[:type:][initial-value]}
in
```

```
        record
            {method-name = makeWrapper[abs [[x1,...,xn]] = body end],}
        end
    end
```

For example, the object

```
    obj
        var a:Int = 0
        method read[[]] =
            pr["\nThe value of a is:"];
            prInt[_a]
        end
        method write[[b]] =
            a <- _a + b;
            pr["\nThe value of a is:"];
            prInt[_a]
        end
    endobj
```

defines one instance variable and two methods. The value of an instance variable can be retrieved by pre-pending an underscore to the variable name and can be set using the `<-` operator. This expression translates to

```
    let
        val a = ref[:Int:][0]
    in
        record
            read =      makeWrapper[
                            abs [[]] =
                                pr["\nThe value of a is:"];
                                prInt[a.deref[]]
                            end],
            writer =    makeWrapper[
                            abs [[b]] =
                                a <- a.deref[] + b;
                                pr["\nThe value of a is:"];
                                prInt[a.deref[]]
                            end]
        end
    end
```

The predefined polymorphic function `ref` creates a new mutable cell like the one shown in §3.5. The only difference is that we use `deref` instead of `get`. Remark that an instance variable reference `_`*a-name* is transformed into *a-name*`.deref[]`. The polymorphic operator `<-` is defined as follows:

```
    def (<-) [:X:][cell: Ref X, value:X] = cell.set[value]
```

where `Ref X` is the type of a mutable cell containing values of type `X`.

`makeWrapper` takes a method and wraps it within a synchronization wrapper as described in §4.1.1. Actually `makeWrapper` does not simply return a new process abstraction representing the wrapped method but a record containing 1) a method wrapped in a empty synchronization wrapper 2) a function to set a new synchronization wrapper. Such an encapsulation is necessary

because when we create the object we still do not know which policy it will be bound to (which is the main difference with McHale's thesis). On the other hand, this allows us to change the synchronization of an object at run-time. The type of the value returned by `makeWrapper` is

```
type MethodWrapper = Fun (X)
        Record
            SetWrappers : ![PreWrapperType,PostWrapperType,Sig],
            WrappedMethod : ![X,Sig]
        end
```

This definition deserves some explanation: we can see that `MethodWrapper` is parameterized by a type `X` which represents the parameters' type of the wrapped method. One can wonder why the type of the wrapped method (which should be the same as the original method) is `![X,Sig]` instead of just `!X`. The reason is the following: to wrap a method in a pre- and post-synchronization code we should be able to put it in a sequence. This means that a method has to be *sequential*, i.e its type has to be $![X_1,...,X_n,Sig]$ (see 3.4.4) where X_i represents the type of the i-th parameter. At first sight it seems that the type `![X,Sig]` expresses just that kind of method. Unfortunately, this is not quite true because `![X,Sig]` only matches a pair and not a tuple of any length (i.e. any number of parameters). We circumvent this problem by requiring that the type of a method be

$$![[X_1,...,X_n],Sig]$$

instead of

$$![X_1,...,X_n,Sig]$$

This explains the "special" syntax (i.e. the double brackets) for method parameters.

The function `SetWrappers` takes a pre-synchronization function `PreWrapper` and a post-synchronization function `PostWrapper` and wraps the method with them. Here are the definitions of `PreWrapperType` and `PostWrapperType`

```
type PreWrapperType = ![!InvocationType,Sig]
type PostWrapperType = ![?InvocationType,Sig]
```

`InvocationType` is the type of the *state* variable mentioned in §4.1.1.

It is now time to present the `makeWrapper` function

```
def makeWrapper [:X:][m:![X,Sig],r:!(MethodWrapper X)] >
let
    new wrappers: ^[PreWrapperType,PostWrapperType]
in
    wrappers![abs [invr] = [] end,abs [invr] = [] end]
|   r!record
        SetWrappers =
            abs [w1:PreWrapperType,w2:PostWrapperType,r:Sig] >
                wrappers?[_,_] > (wrappers![w1,w2] | r![])
            end,
        WrappedMethod =
            abs [p:X,r:Sig] >
                wrappers?[pre_w,post_w] >
                    (   wrappers![pre_w,post_w]
                    |   let
                            new invr : ^InvocationType
```

```
                              in
                                  pre_w[invr];m[p];post_w![invr,r]
                              end
                         )
               end
     end
end
```

The channel `wrappers` contains the current pre- and post-synchronization functions of the method. Initially these correspond to `abs [invr] = [] end`. `SetWrappers` has been implemented in the same way as the `set` method of our mutable cell. More interesting is the implementation of `WrappedMethod`. It works as follows; first we fetch the two current wrappers (as in the `get` method of our mutable cell) and then we call the pre_synchronization `pre_w`. Once `pre_w` has returned we can start executing the method followed by the post-synchronization code. The channel `invr` will be bound to an invocation object by `pre_w` and used by `post_w`.

To close the discussion on our object model, we introduce an operator that can be used to call a method of an object:

```
def (%%)[:X:][mw:MethodWrapper X,p:X] = mw.WrappedMethod[p]
```

For example if `o` is bound to the object defined on page 12, then we can call its method `write` like this

```
o.write%%[2]
```

In summary, the object model we have defined so far simply allows us to associate a synchronization wrapper with a method. Moreover this can be done at run-time. In fact, this model can be used with any synchronization mechanism as long as it can be implemented with synchronization wrappers. In the next subsections we will show how a particular synchronization mechanism, GSP, can be expressed in Pict and linked to our object model.

4.2 GSP in Pict: a first implementation

This subsection describes a partial implementation of GSP in Pict. The complete implementation will be presented in §4.3. We hope that this step-by-step presentation will help the reader to understand it more easily.

4.2.1 The shared data structure

The main goal of a Generic Synchronization Policy is to create a synchronization wrapper for each method category. Of course these wrappers do not work in isolation and need to access shared data. Typically, the synchronization counters (e.g. the number of waiting method of a certain category) belong to the data that will be shared by all synchronization wrappers. In the current implementation these data are stored in a record whose declaration reads as follows:

```
type ActionType = ![InvocationType,Sig]
type GuardType  = ![InvocationType,!Bool]
type PolicyType = Record
                    Mutex : ^[],
                    Clock : ^Int,
                    ArrivalCount : Array Int,
                    StartCount : Array Int,
```

```
                TermCount : Array Int,
                Start : Array ActionType,
                Guard : Array GuardType,
                WaitingList : Ref (List InvocationType),
                ExecList : Ref (List InvocationType)
            end
```

`ActionType` is the type of an action while `GuardType` is the type of a guard. `PolicyType` defines all the necessary data needed by the synchronization code:

- `Mutex` is a semaphore used to put the events in series, i.e. to execute the actions and guards in an atomic way.
- `Clock` is the clock local to this policy and is incremented every time a method is called.
- `ArrivalCount`, `StartCount` and `TermCount` represent the different synchronization counters maintained by the policy. `ArrivalCount` is the total number of invocation that have arrived at the object, `StartCount` is the total number of invocation that have started execution and `TermCount` is the total number of invocation that have terminated execution. These counters are arrays indexed by the method category. For example if the category `ReadOps` of the `ReadersWriters` policy is associated with `1` then the number of currently executing invocations from the category `ReadOps` (i.e `exec(ReadOps)`) can be computed by

  `StartCount[1] - TermCount[1]`
- `Start` contains the actions associated with the *start* event and `Guard` contains the guards of the methods. Both arrays are indexed by the method categories too.
- Finally `WaitingList` (resp. `ExecList`) contains a list of all pending (resp. executing) invocations

4.2.2 Generic code for pre- and post- synchronization functions

The code we have to generate in the pre- and post- synchronization functions is almost the same for every policy. It is only parameterized by the actions and method categories. The specification of the synchronization wrappers is as follows (in pseudo-code where the clause generic introduces generic parameters):

```
type Invocation =    record
                         trigger : semaphore init 0;
                         ArrivalTime : Int;
                         Method_Category : Int;
                     end

generic OPS, ARRIVAL_ACTION, POLICY
pre_synch_OPS()
    var inv : Invocation
begin
    P(POLICY.mutex);
        inv := new Invocation(POLICY.Clock++,OPS);
        POLICY.WaitingList.put(inv);
        POLICY.ArrivalCount[OPS]++;
        ARRIVAL_ACTION(inv);
        evaluate_guards(POLICY);
    V(POLICY.mutex);
```

```
        P(inv.trigger);
        return inv;
end pre_synch_OPS;

generic TERM_ACTION, POLICY
post_synch_OPS(inv:Invocation)
begin
        P(POLICY.mutex);
                POLICY.ExecutingList.remove(this_inv);
                POLICY.TermCount[this_inv.Method_Category]++;
                TERM_ACTION(inv);
                evaluate_guards(POLICY);
        V(POLICY.mutex);
end post_synch_OPS;
```

OPS stands for the number associated with the method category, ARRIVAL_ACTION (resp. TERM_ACTION) is the action associated with the *arrival* (resp. *term*) event and finally POLICY represents the data shared by all synchronization wrappers. The algorithm for pre_synch works as follows: first we enter a critical section to access the shared data. Then we create a new invocation object initialized with the local time (that we immediately increment) and the method category. We put this invocation in the list of pending invocations and update the number of arrived invocations. Then we execute the action associated with the *arrival* event. At this time it could be that the guard of one of the pending invocation has become true. We therefore call evaluate_guards() whose purpose is to wake up invocations that are allowed to continue. Finally, we exit the critical section and wait on the semaphore inv.trigger until the corresponding guard becomes true. The result of the function is the newly created invocation which will be passed to the post_synch function. The latter updates the list of executing invocations and the number of terminated invocation. It then executes the action associated with the *term* event and finally calls evaluate_guards(). As guards can only access information local to a given policy, this means that they need only be re-evaluated when this information has been modified, i.e in the functions pre_synch and post_synch. The implementation of evaluate_guards() is the following

```
evaluate_guards(policy:PolicyType)
        var inv:Invocation;
begin
        while findInvocation(policy,inv) do
                V(inv.trigger);
                policy.StartCount[inv.Method_Category]++;
                policy.WaitingList.remove(inv);
                policy.ExecList.put(inv);
                policy.Start[inv.Method_Category](inv);
        end while;
end evaluate_guards;

findInvocation(policy:PolicyType; VAR inv:Invocation) : Boolean
begin
        for i in policy.WaitingList do
                if policy.Guard[i.Method_Category](i) then
                        inv := i;
                        return true;
                end;
```

```
        end;
        return false;
    end findInvocation;
```

findInvocation is a function used to retrieve the first pending invocation whose guard evaluates to true. Given this auxiliary function, the implementation of evaluate_guards is relatively straightforward: as long as there is an invocation ready to run we remove it from the list of waiting invocations and put it into the list of running invocations, update the number of executing invocations and finally execute the associated action. It is important to note that each time we look for another invocation we start from the beginning of the list. This is necessary because the action of the preceding starting invocation may cause other guards to become true (by updating some synchronization variables).

The concrete Pict code is a straightforward translation of the above pseudo-code

```
type InvocationType =   Record
                            Trigger : Sig,
                            ArrivalTime : Int,
                            GenMethRef : Int
                        end
def MakePreWrapper[Pol:PolicyType,GenMethRef:Int,Action:ActionType] =
    abs [invr:!InvocationType,r:Sig] >
        Pol.Mutex?[] >
        Pol.Clock?clock >
        Pol.Clock!(clock+1)
    |   let
            val NewInv : InvocationType =
                            record
                                Trigger = r,
                                ArrivalTime = clock,
                                GenMethRef = GenMethRef
                            end
        in
            Pol.WaitingList.set[
                    cons[NewInv,Pol.WaitingList.deref[]]];
            updateArray[Pol.ArrivalCount, GenMethRef,
                        nthArray[Pol.ArrivalCount,
                        GenMethRef]+1];
            Action[NewInv];EvaluateGuards[Pol];
            (Pol.Mutex![] | invr!NewInv)
        end
    end
def removeInv[l:List InvocationType, time:Int] =
    if null[l] then
        nil[]
    elif (unsafeCar[l]).ArrivalTime == time then
        unsafeCdr[l]
    else
        unsafeCar[l] @@ removeInv[unsafeCdr[l],time]
    end
def MakePostWrapper[Pol:PolicyType, Action:ActionType] =
    abs [invr:?InvocationType,r:Sig] >
        invr?Inv >
```

```
        Pol.Mutex?[] >
        Pol.ExecList.set[removeInv[Pol.ExecList.deref[],
                                  Inv.ArrivalTime]];
        updateArray[Pol.TermCount,Inv.GenMethRef,
                nthArray[Pol.TermCount,Inv.GenMethRef]+1];
        Action[Inv];
        EvaluateGuards[Pol];
        Pol.Mutex![]
    end
```

Note that the generic parameters have been transformed into the formal parameters of a process abstraction. To create a pre-synchronization function we call `MakePreWrapper` with the corresponding policy object, method category and action. A semaphore is simply represented as a channel on which we output an empty tuple. The `P` operation amounts to reading on this channel, i.e, consuming the process *`semaphore-name`*`![]`. The `V` operation is then only a creation of such a process. Arrays are accessed via the two predefined functions `updateArray[`*`array`*`,`*`index`*`,`*`newvalue`*`]` and `nthArray[`*`array`*`,`*`index`*`]`. The only subtle point here is the trigger used to suspend the method. Actually what will be stored as a semaphore in the invocation object is the "continuation" channel of the `pre_synch` function. This means that sending an empty tuple on this channel will allow the statement following the call to `pre_synch` to start executing, which corresponds to our method's body. The function `removeInv` is used to remove invocations from the waiting list. The head of the list can be retrieved with `unsafeCar` and the tail with `unsafeCdr`. A new list can be constructed from a head and a tail with the operator `@@`. As we cannot use pointers for testing invocation equality we have to use the arrival time of the invocation.

```
def EvaluateGuards[Pol:PolicyType, r:Sig] >
    let
        new inv, found
        def findInvocation[l:List InvocationType,
                            r:!(List InvocationType)] >
            if null[l] then
                found!false | nil![r]
            else
                let
                    val i = unsafeCar[l]
                in
                    if (nthArray[Pol.Guard, i.GenMethRef])[i]
                    then
                        found!true | inv!i | unsafeCdr![l,r]
                    else
                        cons![i, findInvocation[unsafeCdr[l]],r]
                    end
                end
            end
        def while[] >
            Pol.WaitingList.set[
                    findInvocation[Pol.WaitingList.deref[]]];
            (found?b >
            if b then
                inv?i > i.Trigger![] |
                (updateArray[Pol.StartCount, i.GenMethRef,
```

```
                    nthArray[Pol.StartCount, i.GenMethRef]+1];
                Pol.ExecList.set[cons[i, Pol.ExecList.deref[]]];
                (nthArray[Pol.Start, i.GenMethRef])[i];
                (while![]))
            else
                r![]
            end)
    in
        while![]
    end
```

As in the pseudo-code the function `findInvocation` retrieves the first invocation for which the guard evaluates to true. The only difference here is that the invocation is immediately removed when found. This allows us to save a second scan of the list. The function therefore returns the new list and, as a side-effect, updates the variable `found` which tells if an invocation has been found or not. It the answer is yes then the invocation is stored in the `inv` variable (`cons` is just a synonym for `@@`). The `while` process implements a simple while loop.

The Pict code presented so far constitutes the run-time support for GSP and is provided as a file that must be included in every program using this synchronization mechanism. The remainder of the section introduces the syntax chosen to express the policy itself, the pre-processing done to this syntax, and the way to bind such a policy to an object.

4.2.3 Syntax of GSP in Pict

The syntax of a Generic Synchronization Policy expressed in Pict is the following (again this syntax will be treated by the preprocessor):

```
policy [method-category1,...,method-categoryN]
    local-pict-declarations
    map {event-name-or-guard // [ method-category,function-name]; }
end
```

The *`local-pict-declarations`* part define the synchronization variables, actions and guards that will be used in the map section. This latter corresponds to the map section introduced in §2.2. *`event-name-or-guard`* stands for one of the four following names: `Arrival`, `Start`, `Term` and `Guard`. For example, the *Reader-Writer* policy example is expressed as follows

```
val ReadersWriters = policy [ReadOps,WriteOps]
                        def ReaderAllowed[t:InvocationType] =
                            exec[WriteOps] == 0
                        def WriterAllowed[t:InvocationType] =
                                exec[ReadOps]+exec[WriteOps] == 0
                        map Guard//[ReadOps,ReaderAllowed];
                            Guard//[WriteOps,WriterAllowed]
                        end
```

`ReadersWriters` will then be instantiated to yield a synchronization policy that can be bound to an object

```
val mypolicy = ReadersWriters[]
run BindPolicy[p.ReadOps**o.read + p.WriteOps**o.write]
```

Here `o` corresponds to the object defined in §4.2.1. The expression `BindPolicy[ p.ReadOps**o.read + p.WriteOps**o.write]` binds the `read` method to the `ReadOps` category and the method `write` to the `WritOps` category. Actually, `mypolicy` is a record with two fields `ReadOps` and `WritOps` containing the synchronization wrappers. More generally, the result of instantiating a policy of the form

```
policy [method-category1,...,method-categoryN]
 .....
end
```

is a record of type

```
Record
    method-category1: PrePostWrappers,
    ...
    method-categoryN: PrePostWrappers
end
```

with `PrePostWrappers` defined as

```
type PrePostWrappers =  Record
                            preWrapper : PreWrapperType,
                            postWrapper: PostWrapperType
                        end
```

It follows that the `**` and `BindPolicy` functions are trivially defined as

```
def BindPolicy[n:Int] = []
def (**)[:X:][ppWrapper:PrePostWrappers,
                mWrapper:MethodWrapper X,r:!Int] >
            mWrapper.SetWrappers[ppWrapper.preWrapper,
                                ppWrapper.postWrapper];
            r!0
```

Thus, the goal of the preprocessing phase is to produce a function that return such a record using the run-time functions introduced in §4.2.2. Here it is:

```
policy [method-category1,...,method-categoryN]
local-pict-declarations
map mapping-declarations
end
```

translates to

```
let
    val nbmeth=makeCounter[0]
    val method-category1 = nbmeth.incr[]
    ...
    val method-categoryN = nbmeth.incr[]
in
    abs[] = let
        val Start = makeArray[EmptyAction,nbmeth.value[]]
        val Arrival = makeArray[EmptyAction,nbmeth.value[]]
        val Term = makeArray[EmptyAction,nbmeth.value[]]
        val Guard = makeArray[EmptyGuard,nbmeth.value[]]
        val pol = makePolicy[Start,Guard,nbmeth.value[]]
        def exec[n:Int] =
```

```
            nthArray[pol.StartCount,n]-nthArray[pol.TermCount,n]
        def waiting[n:Int] =
            nthArray[pol.ArrivalCount,n]-
                    nthArray[pol.StartCount,n]
        def waitingList[n:Int]=
                extractInvocation[pol.WaitingList.deref[],n]
        local-pict-declarations
        val [] = mapping-declarations
    in
        record
        method-category1 =
            makeSynchWrappers[Arrival,Term,pol,method-category1],
            ...
        method-categoryN =
            makeSynchWrappers[Arrival,Term,pol,method-categoryN]
            end
        end
    end
end
```

Basically, the only thing we do is to include the policy definitions, i.e. the `local-pict-declarations` and `mapping-declarations` parts, into some definitions. These are actually the so-called "predefined" names (or keywords) of the GSP concept: `Start`, `Arrival`, `Term`, `Guard`, `exec`, `waiting`, and `waitingList`. The first four keywords are references to arrays that contains the actions and guards associated with each method category. This allows us to treat a mapping declaration of the form

```
event-name-or-guard // [method-category,function-name];
```

as an already correct Pict expression. The trick here is to define `//` as follows

```
def (//)[:X:][a:Array X,[n:Int,x:X]] = updateArray[a,n,x]
```

Of course, *`method-category`* must refer to a natural number. This binding is done in the outermost `let` construct:

```
    val nbmeth=makeCounter[0]
    val method-category1 = nbmeth.incr[]
    ...
    val method-categoryN = nbmeth.incr[]
```

The function `makeCounter` simply returns a counter that can be read and incremented

```
def makeCounter[n:Int] =
        let new c
            run c!n
        in
            record
                incr    = abs [r:!Int] > c?n > (c!(n+1) | r!n) end,
                value   = abs [r:!Int] > c?n > (c!n | r!n) end
            end
        end
```

Instead of using a counter, we could have directly bound a fixed number to a method category during the preprocessing phase but we wanted to keep this phase as simple as possible.

Finally, once the different arrays have been updated, we can build the record containing the synchronization wrappers. For this purpose, we first have to create a new record `pol` of type `PolicyType` that we then pass as a parameter to the `makeSynchWrappers` function defined as follows:

```
def makeSynchWrappers[Arrival:Array ActionType,
                      Term:Array ActionType,
                      pol:PolicyType,n:Int] =
    record
        preWrapper=MakePreWrapper[pol,n,nthArray[Arrival,n]],
        postWrapper=MakePostWrapper[pol,nthArray[Term,n]]
    end
```

For the sake of completeness, here are the remaining definitions of `EmptyAction`, `EmptyGuard`, `makePolicy` and `extractInvocation`:

```
def EmptyAction[inv:InvocationType] = []
def EmptyGuard[inv:InvocationType] = true

def makePolicy[Start:Array ActionType,Guard:Array GuardType,n:Int] =
    let
        new Mutex : ^[], Clock : ^Int
        run Clock!0 | Mutex![]
    in
        record
            Mutex = Mutex,
            Clock = Clock,
            ArrivalCount = makeArray[0,n],
            StartCount = makeArray[0,n],
            TermCount = makeArray[0,n],
            Start = Start,
            Guard = Guard,
            WaitingList = ref[nil[]],
            ExecList = ref[nil[]]
        end
    end

def extractInvocation[l:List InvocationType,GenMethRef:Int] =
    if null[l] then
        nil[]
    elif (unsafeCar[l]).GenMethRef <> GenMethRef then
        extractInvocation[unsafeCdr[l],GenMethRef]
    else
        unsafeCar[l] @@ extractInvocation[unsafeCdr[l],GenMethRef]
    end
```

4.3 Accessing the parameters of a method

The code shown so far is not a complete implementation of the GSP mechanism. Actually, to be able to express some interesting synchronizations we also need access to the parameters of the method. Consider for example the Reader-Writer policy of section 2.2 and assume we want the writers to be scheduled according to their jobs' size, that is we want to schedule the shortest job first. We suppose that the job's size is a parameter of the methods associated with WriteOps. An implementation of this policy could look this

```
policy ReadersWriters[ReadOps,WriteOps[len:Int]]
function ReaderAllowed(t:Invocation):Bool
begin
    return exec(WriteOps) = 0;
end
function ShortestJob(len:Int):Bool
begin
    for p in waiting(WriteOps) do
        if p.len < len then
            return false;
        end
    end
    return true;
end
function WriterAllowed(t:LenInvocation):Bool
begin
    return exec(ReadOps)+exec(WriteOps) = 0 and
        ShortestJob(t.len);
end
map guard(ReadOps) -> ReaderAllowed
    guard(WriteOps) -> WriterAllowed
end policy
```

The name and type of the parameters are specified in the policy's signature. The value of the corresponding actual parameters are stored in fields of the invocation object and can then be retrieved using the dot notation. Note that we now have two classes of invocations: one for the ReadOps category with a len field and one for the WriteOps with no additional field. This raises an interesting type question that will be dealt with in the next subsections together with the necessary modifications to our current implementation. We will also explain how to specify the correspondence between the formal parameters of a method category and the associated methods.

4.3.1 Storing and retrieving the parameters

In McHale's thesis, an invocation's class associated with a method category like `WriteOps` is just defined as a subclass of `InvocationType` with a new field for each formal parameter. We decided, in our implementation, to store all the parameters in one field, in the form of a tuple of values (or as a single value if there is only one parameter). This gives us the following type definitions:

```
type InvocationType =   Record
                            Trigger : Sig,
                            ArrivalTime : Int,
                            GenMethRef : Int,
                        end
type Invocation =       Fun(X) InvocationType
                            with parameters : X end
```

The base type is `InvocationType` defined as before. `Invocation` is a type operator that takes a type `X` and returns an extended `InvocationType` record with a field `parameters` of type `X`. For example the type of invocations associated with `WriteOps` would be:

```
type WriteOpsInvocation = Invocation Int
```

We can then use this additional field to retrieve the parameter's value in the code for actions and guards. This implies that if a method category defines parameters of type `X` then the actions and guard associated with it must take as parameter a value of type `Invocation X`. This apparently harmless constraint raises an interesting type problem. To see it consider the lifetime of an invocation: when created, an invocation of type `Invocation X` is passed to the *arrival* action, then put into a list of waiting invocations. Once its guard evaluates to true it is passed to the *start* action and finally, after the method's completion, to the *term* action. Now the question is: what is the type of the list of waiting invocations? The type of its elements must be a supertype of all possible invocation types in order for them to be stored in the same data structure, which means that our list will have the same type as before, i.e. `List InvocationType`. Unfortunately, this implies that when we extract an invocation from the list its field `parameters` cannot be accessed any more because this would require a "downcasting" to the actual type of the invocation, which is not possible in Pict. Thus, we cannot pass an invocation of the right type to the guard and the *start* and *term* actions.

One solution to this problem is to give up the idea of a unique list of invocations. Thus, we would keep a separate list of invocations for each method category, so that we could define it with a more precise type. For example the type of the list associated with `WriteOps` would be `List (Invocation Int)`. In this way we do not lose any type information at all, which means that no downcasting is necessary. But this also means that we have to manage these different lists in our generic code. For example a new `PolicyType` should be defined for each policy because another policy would typically have a different number of waiting lists with different types. The same would be true for the `EvaluateGuards` function. In other words, this means that a big part of the generic code would not be generic any more and should be included in the preprocessing phase of our implementation, which is exactly what we are trying to avoid. Therefore, we gave up this solution although we will see in the next section that we still need to keep local copies to implement the `waitingList` function.

To see how we can solve this problem while keeping a unique list, we need to point out exactly where the problem lies. Reconsidering the lifetime of an invocation, we immediately see that the difficulty comes from the fact that the binding of an action (or a guard) to its actual parameter (which is an invocation object) occurs when we have already lost information about the parameter. Therefore, the solution is simply to do this binding *before* any information gets lost, in other words before the invocation is put in the waiting list. In order to do this we need to separate the binding of an action to its actual parameter from the call of the action itself (these two events usually take place at the same time) because, for example, we do not want to call the *term* action before the invocation is put in the list. Therefore, we will transform a function call of the form

```
Action[InvocationRecord]
```

into

```
def ActionTrigger[] = Action[InvocationRecord]
... and some time later ...
ActionTrigger[]
```

The binding of the function to its parameter is done during the definition of `ActionTrigger` while the actual call is done by `ActionTrigger[]`. Of course, now for each pair (*Action*,*InvocationRecord*) we have a different `ActionTrigger` that we need to remember. The obvious so-

lution is to put those triggers in the invocation record itself so that we come to following and final type definitions (the modified code is written in underlined Courier)

```
type SimpleAction = ![Sig]
type SimpleGuard = ![!Bool]
type InvocationType =   Record
                            Trigger : Sig,
                            ArrivalTime : Int,
                            GenMethRef : Int,
                            StartAction : SimpleAction,
                            TermAction : SimpleAction,
                            Guard : SimpleGuard
                        end
type Invocation = Fun(X) InvocationType
                        with parameters : X end
```

where the field `StartAction` holds the trigger for the *start* action (idem for `TermAction` and `Guard`). The triggers are created together with the invocation object in the pre-synchronization code of the method:

```
type ActionType = Fun(X) ![Invocation X,Sig]
type GuardType  = Fun(X) ![Invocation X,!Bool]
def MakePreWrapper [:X:][Pol:PolicyType,GenMethRef:Int,
                        ArrivalAction:ActionType X,
                        StartAction:ActionType X,
                        TermAction:ActionType X,
                        Guard:GuardType X] =
    abs [invr:!InvocationType,param:X,r:Sig] >
        Pol.Mutex?[] >
        Pol.Clock?clock >
        Pol.Clock!(clock+1) |
    let
        val self = emptyRef[:Invocation X:][]
        val NewInv : Invocation X =
                record
                    Trigger = r,
                    ArrivalTime = clock,
                    GenMethRef = GenMethRef,
                    StartAction =
                        abs[] = StartAction[self.deref[]] end,
                    TermAction =
                        abs[] = TermAction[self.deref[]] end,
                    Guard = abs[] = Guard[self.deref[]] end,
                    parameters = param
                end
    in
        self.set[NewInv];
        Pol.WaitingList.set[cons[:InvocationType:]
                                [NewInv, Pol.WaitingList.deref[]]
                            ];
        updateArray[Pol.ArrivalCount,GenMethRef,
                nthArray[Pol.ArrivalCount,GenMethRef]+1];
        ArrivalAction[NewInv];
        EvaluateGuards[Pol];
```

```
        (Pol.Mutex![] | invr!NewInv)
        end
    end
```

This function is the one on page 13 modified as follows:

- it accepts three more parameters StartAction, TermAction, Guard necessary to create the triggers. Moreover, it has been made polymorphic to take the different invocation types into account.
- the generated pre-synchronization function takes one more parameter param which corresponds to the parameters of the associated method category. More will be said about it in the next sub-section.
- the invocation record is of type `Invocation X` and not `InvocationType` and is initialized accordingly. Here we need to use an additional indirection `self` to reference the invocation record in the triggers because we cannot directly write something like:

```
val x = record
            ActionTrigger = abs [] = Action[x] end
        end
```

Actually, we can do it without any indirection but then the syntax is less readable. The idea is simply to come back to the core Pict where an abstraction like:

```
abs [] = Action[x] end
```

translates to

```
let
    new a
    run a?*[] = Action[x]
in
    a
end
```

We can then transform

```
val x = record
            ActionTrigger = abs [] = Action[x] end
        end
```

into

```
new a
val x = record
            ActionTrigger = a
        end
run a?*[] = Action[x]
```

The other two functions `EvaluateGuards` and `MakePostWrapper` are modified as follows

```
def EvaluateGuards[Pol:PolicyType,r:Sig] >
    let
        new inv, found
        def findInvocation[l:List InvocationType,
                                    r:!(List InvocationType)] >
            if null[l] then
                found!false | nil![r]
```

```
                else
                    let
                        val i = unsafeCar[l]
                    in
                        if i.Guard[] then
                            found!true | inv!i | unsafeCdr![l,r]
                        else
                            cons![i,findInvocation[unsafeCdr[l]],r]
                        end
                    end
                end

            def while[] >
                Pol.WaitingList.set[
                        findInvocation[Pol.WaitingList.deref[]]];
                (found?b >
                if b then
                    inv?i > i.Trigger![] |
                    (updateArray[Pol.StartCount,i.GenMethRef,
                        nthArray[Pol.StartCount,i.GenMethRef]+1];
                    Pol.ExecList.set[cons[i,Pol.ExecList.deref[]]];
                    i.StartAction[];
                    (while![]))
                else
                    r![]
                end)
        in
            while![]
        end

    def MakePostWrapper[Pol:PolicyType] =
        abs [invr:?InvocationType,r:Sig] >
            invr?Inv >
            Pol.Mutex?[] >
            Pol.ExecList.set[removeInv[Pol.ExecList.deref[],
                                        Inv.ArrivalTime]];
            updateArray[Pol.TermCount,Inv.GenMethRef,
                        nthArray[Pol.TermCount,Inv.GenMethRef]+1];
            Inv.TermAction[];EvaluateGuards[Pol];Pol.Mutex![]
        end
```

Here we just changed the way actions and guards are called. As a consequence we can simplify the definition of `PolicyType` to

```
    type PolicyType =   Record
                            Mutex : ^[],
                            Clock : ^Int,
                            ArrivalCount : Array Int,
                            StartCount : Array Int,
                            TermCount : Array Int,
                            WaitingList : Ref (List InvocationType),
                            ExecList : Ref (List InvocationType)
                        end
```

that is, we can drop the fields `Start` and `Guard` (`makePolicy` is changed accordingly)

4.3.2 Mapping method parameters

Until now we have only concentrated on the modification of the policy's functions. From the synchronized object's point of view the situation also gets more complicated. To see this, consider the following object definition

```
obj
    var a:Int = 0
    method read[[]] =
        pr["\nThe value of a is:"];
        prInt[_a]
    end
    method write[[b,c]] =
        a <- _a + b + c;
        pr["\nThe value of a is:"];
        prInt[_a]
    end
endobj
```

and assume we want to synchronize this object with the policy on page 23, binding `read` to `ReadOps` and `write` to `WriteOps`. For the second method we have to specify which of the two parameters `b` and `c` will be mapped to the formal parameter of `WriteOps`. We cannot do it when we create the wrapped method (using `makeWrapper`) because at that time we still don't know to which policy the object will be bound. So we do it when calling the `BindPolicy` function. We opted for a solution where we specify this mapping by a function called an *extractor*. An extractor simply returns a subset of the parameters of a method as a tuple of values (or as a single value if there is only one parameter). For example if we want to bind the first parameter `b` to the formal parameter of `WriteOps` we write it like this:

```
BindPolicy[p.ReadOps**[o.read,abs [[]] =[] end]
        + p.WriteOps**[o.write,abs [[b,c]] = b end]]
```

The abstraction `abs [[]] =[] end` represents the extractor for the `read` method (i.e an empty tuple is used for a parameterless method category) and `abs [[b,c]] = b end` the extractor for the `write` method. Had we decided to bind `c` to the formal parameter of `WriteOps`, then we would have written it as follows

```
BindPolicy[p.ReadOps**[o.read,abs [[]] =[] end]
        + p.WriteOps**[o.write,abs [[b,c]] = c end]]
```

To take this parameter's mapping into account we need to slightly modify the function `makeWrapper` and the related type definitions:

```
type PreWrapperType = Fun(X) ![!InvocationType,X,Sig]
type PostWrapperType = ![?InvocationType,Sig]
type ParamExtractorType = Fun(X) Fun(Y) ![X,!Y]

type MethodWrapper =Fun (X) Record
        SetWrappers : ![:Y:][PreWrapperType Y,PostWrapperType,
                             ParamExtractorType X Y,Sig],
        WrappedMethod : ![X,Sig]
        end
```

```
def makeWrapper [:X:][m:![X,Sig],r:!(MethodWrapper X)] >
    let new wrappers: ^[PreWrapperType X,PostWrapperType] in
        wrappers![abs [invr,X] = [] end,abs [invr] = [] end]
    |   r!
        record
            SetWrappers =
                abs[:Y:][new_pre_w:PreWrapperType Y,
                    new_post_w:PostWrapperType,
                    paramExtractor:ParamExtractorType X Y,r:Sig] >
                    wrappers?[_,_] >
                    (wrappers![abs [invr,X] =
                            new_pre_w[invr,paramExtractor[X]]
                            end,
                            new_post_w] | r![])
                end,
            WrappedMethod =
                abs [p:X,r:Sig] >
                    wrappers?[pre_w,post_w] >
                    (   wrappers![pre_w,post_w]
                    |   let
                            new invr : ^InvocationType
                        in
                            pre_w[invr,p];m[p];post_w![invr,r]
                        end
                    )
                end
        end
end
```

This definition deserves some comments: `PreWrapperType` has been modified to reflect the fact that a pre-synchronization wrapper generated by `makeWrapper` expects one more argument corresponding to the formal parameters of the method category. `SetWrappers` has also been modified and now expects two wrappers and one extractor. The type parameter `X` stands for the type of the method's arguments while `Y` stands for the type of the method category's parameters. More interesting is the implementation of `WrappedMethod`. At first one could think of implementing it like this:

```
abs [p:X,r:Sig] >
    wrappers?[pre_w,post_w,extractor] >
        (   wrappers![pre_w,post_w,extractor]
        |   let
                new invr : ^InvocationType
            in
                pre_w[invr,extractor[p]];m[p];post_w![invr,r]
            end
        )
```

But this solution does not work because this would imply that the type `Y` is already known when the wrapped method is built, which is in contradiction with the fact that a policy is bound to an object later on. We therefore have to hide this type with a wrapper's wrapper that will be generated by the expression

```
abs [invr,X] = new_pre_w[invr,paramExtractor[X]] end
```

whenever the `SetWrappers` function is called.

4.3.3 Preprocessing

In the preprocessing phase described on page 20 we used four arrays to hold the actions and guard associated with the different method categories. This solution is no longer possible because actions and guards do not have the same type for all method categories. Therefore, we chose to group the three actions and the guard of a method category in a record of the following type:

```
type CategoryInfo = Fun (X)
                    Record
                        GenMethRef : Int,
                        ArrivalAction : Ref (ActionType X),
                        StartAction : Ref (ActionType X),
                        TermAction : Ref (ActionType X),
                        Guard : Ref (GuardType X)
                    end
```

Because the `Arrival`, `Start`, `Term` and `Guard` names do not refer to arrays any more we have to find another definition for them. Actually we decided to make a slight change to the syntax, transforming

```
event-name-or-guard // [ method-category,function-name]
```

into

```
event-name-or-guard [ method-category,function-name]
```

that is, we just dropped the `//` operator that is no longer necessary. This gave us the following definitions for our four names

```
def Arrival[:X:][cat:CategoryInfo X,Action : ActionType X] =
                    cat.ArrivalAction.set[Action]
def Start[:X:][cat:CategoryInfo X,Action : ActionType X] =
                    cat.StartAction.set[Action]
def Term[:X:][cat:CategoryInfo X,Action : ActionType X] =
                    cat.TermAction.set[Action]
def Guard[:X:][cat:CategoryInfo X,Guard: GuardType X] =
                    cat.Guard.set[Guard]
```

From the type of these functions we can infer that a method category name will not be bound to a number but to a record of type `CategoryInfo X` where `X` is the type of the method category's formal parameters. The other difference with the previous preprocessing phase is that `Arrival`, `Start`, `Term` and `Guard` are now globally defined. Given these definitions a policy declaration of the form

```
policy [method-category1,...,method-categoryN]
local-pict-declarations
map mapping-declarations
end
```

translates to

```
abs[] =
    let
        val nbmeth=makeCounter[0]
```

```
        val method-category1 = makeCategoryInfo[nbmeth.incr[]]
        ...
        val method-categoryN = makeCategoryInfo[nbmeth.incr[]]

        val pol = makePolicy[nbmeth.value[]]
        def exec[:X:][cat:CategoryInfo X] =
            nthArray[pol.StartCount,cat.GenMethRef]-
            nthArray[pol.TermCount,cat.GenMethRef]
        def waiting[:X:][cat:CategoryInfo X] =
            nthArray[pol.ArrivalCount,cat.GenMethRef]-
            nthArray[pol.StartCount,cat.GenMethRef]
        def waitingList[:X:][cat:CategoryInfo X]=
            extractInvocation[pol.WaitingList.deref[],
                                cat.GenMethRef]
        local-pict-declarations
    val [] = mapping-declarations
    in
        record
        method-category1 =
            makeSynchWrappers[pol,method-category1],
            ...
        method-categoryN =
            makeSynchWrappers[pol,method-categoryN]
        end
    end
 end
```

The exec, waiting and waitingList functions have changed accordingly. Here are the remaining definitions:

```
def makeCategoryInfo[:X:][n:Int] =
    let
        val emptyAction = abs [i:X] = [] end
    in
        record
            GenMethRef = n,
            ArrivalAction = ref[emptyAction],
            StartAction = ref[emptyAction],
            TermAction = ref[emptyAction],
            Guard = ref[abs [i:X] = true end]
        end
    end

def makeSynchWrappers[:X:][pol:PolicyType,cat:CategoryInfo X] =
        record
            preWrapper=MakePreWrapper[pol,
                                cat.GenMethRef,
                                cat.ArrivalAction.deref[],
                                cat.StartAction.deref[],
                                cat.TermAction.deref[],
                                cat.Guard.deref[]],
            postWrapper=MakePostWrapper[pol]
        end
```

4.3.4 Getting it right

Consider the policy on page 23 and its translation into Pict

```
policy [Reader,Writer]
def shortestJob[l: List (Invocation Int),len:Int] =
    inFoldList[l,abs[x,res] = res && (x.parameters >= len) end,true]
def ReaderAllowed[i:SimpleInvocation] = exec[Writer]==0
def WriterAllowed[i:Invocation Int] =
    exec[Reader]+exec[Writer]==0 &&
        shortestJob[waitingList[Writer],i.parameters]
map Guard[Reader,ReaderAllowed];
    Guard[Writer,WriterAllowed]
end
```

The function `inFoldList` takes a list $(x_1,x_2,...,x_n)$, a function `f` and an initial value `v`. The result is $f(x_n,f(.....f(x_2,f(x_1,v))))$. Thus, `shortestJob` returns true if `len` is the length of the shortest job in `l`.

At first sight, this program seems correct, but in fact it is not. The problem lies in the type of the list returned by the expression `waitingList[Writer]` which is `List (InvocationType)` instead of `List (Invocation Int)`. This is due to the fact that `waitingList` builds its result from the waiting list stored in the policy and whose type is `List (InvocationType)`. The problem here is always the same: we cannot recover the field `parameters` from a record of type `InvocationType`. Although it may seem that we haven't really solved the problem yet, this impression is not true. The above program can be transformed as follows:

```
policy [Reader,Writer]
val jobList = ref[nil[]]
def putJob[len:Int] = jobList.set[len @@ jobList.deref[]]
def remove[l: List Int,x:Int] =
    if unsafeCar[l] == x then
        unsafeCdr[l]
    else
        unsafeCar[l] @@ remove[unsafeCdr[l],x]
    end
def removeJob[len:Int] = jobList.set[remove[jobList.deref[],len]]
def shortestJob[len:Int] =  inFoldList[jobList.deref[],
                                abs[x,res] = res && (x >= len) end,
                                true
                            ]
def ReaderAllowed[i:SimpleInvocation] = exec[Writer]==0
def WriterAllowed[i:Invocation Int] =
    exec[Reader]+exec[Writer]==0 && shortestJob[i.parameters]
def WriterArrival[i:Invocation Int] = putJob[i.parameters]
def WriterStart[i:Invocation Int] = removeJob[i.parameters]
map Guard[Reader,ReaderAllowed];
    Arrival[Writer,WriterArrival];
    Start[Writer,WriterStart];
    Guard[Writer,WriterAllowed]
end
```

Here the main idea is to keep a separate list of jobs and update it at the events *arrival* and *start*. Having to maintain such a list seems to be a drawback, but in fact we could take advantage of it, for example by keeping the list sorted which would speed up the evaluation of the guards.

Fortunately, it is easy to modify our implementation to allow the `waitingList` to return a value of the right type. For this we need to keep one separate list for each method category that will be update at the same time as the main waiting list in the policy record. Here are the modified functions and types:

```
type InvocationType =   Record
                            Trigger : Sig,
                            ArrivalTime : Int,
                            GenMethRef : Int,
                            StartAction : SimpleAction,
                            TermAction : SimpleAction,
                            RemoveYourSelf : SimpleAction,
                            Guard : SimpleGuard
                        end
def removeInv[:X<:InvocationType:][l:List X, time:Int] =
    if null[l] then
        nil[]
    elif (unsafeCar[l]).ArrivalTime == time then
        unsafeCdr[l]
    else
        unsafeCar[l] @@ removeInv[unsafeCdr[l],time]
    end
def EvaluateGuards[Pol:PolicyType,r:Sig] >
    let
        new inv, found
        def findInvocation[l:List InvocationType,
                                r:!(List InvocationType)] >
            if null[l] then
                found!false | nil![r]
            else
                let
                    val i = unsafeCar[l]
                in
                    if i.Guard[] then
                        i.RemoveYourSelf[];
                        (found!true | inv!i | unsafeCdr![l,r])
                    else
                        cons![i,findInvocation[unsafeCdr[l]],r]
                    end
                end
            end
        def while[] >
            Pol.WaitingList.set[
                        findInvocation[Pol.WaitingList.deref[]]];
            (found?b >
            if b then
                inv?i > i.Trigger![] |
                (updateArray[Pol.StartCount,i.GenMethRef,
                    nthArray[Pol.StartCount,i.GenMethRef]+1];
```

```
                Pol.ExecList.set[cons[i,Pol.ExecList.deref[]]];
                i.StartAction[];
                (while![]))
            else
                r![]
            end)
    in
        while![]
    end

def MakePreWrapper [:X:][Pol:PolicyType,GenMethRef:Int,
                    ArrivalAction:ActionType X,
                    StartAction:ActionType X,
                    TermAction:ActionType X,
                    Guard:GuardType X,
                    waitingList:Ref (List (Invocation X))] =
    abs [invr:!InvocationType,param:X,r:Sig] >
        Pol.Mutex?[] >
        Pol.Clock?clock >
        Pol.Clock!(clock+1) |
    let
        val self = emptyRef[:Invocation X:][]
        val NewInv : Invocation X =
                record
                    Trigger = r,
                    ArrivalTime = clock,
                    GenMethRef = GenMethRef,
                    StartAction =
                        abs[] = StartAction[self.deref[]] end,
                    TermAction =
                        abs[] = TermAction[self.deref[]] end,
                    RemoveYourSelf =
                        abs [] =
                            waitingList.set[
                                removeInv[waitingList.deref[],
                                            clock]]
                        end,
                    Guard = abs[] = Guard[self.deref[]] end,
                    parameters = param
                end
    in
        self.set[NewInv];
        Pol.WaitingList.set[cons[:InvocationType:][NewInv,
                                Pol.WaitingList.deref[]]];
        waitingList.set[cons[NewInv,waitingList.deref[]]];
        updateArray[Pol.ArrivalCount,GenMethRef,
                nthArray[Pol.ArrivalCount,GenMethRef]+1];
        ArrivalAction[NewInv];EvaluateGuards[Pol];
        (Pol.Mutex![] | invr!NewInv)
        end
    end

type CategoryInfo = Fun (X)
        Record
            GenMethRef : Int,
```

```
                ArrivalAction : Ref (ActionType X),
                StartAction : Ref (ActionType X),
                TermAction : Ref (ActionType X),
                Guard : Ref (GuardType X),
                waitingList : Ref (List (Invocation X))
            end

def makeCategoryInfo[:X:][n:Int] =
    let
        val emptyAction = abs [i:X] = [] end
    in
        record
            GenMethRef = n,
            ArrivalAction = ref[emptyAction],
            StartAction = ref[emptyAction],
            TermAction = ref[emptyAction],
            Guard = ref[abs [i:X] = true end],
            waitingList = ref[nil[:X:][]]
        end
    end

def waitingList[:X:][cat:CategoryInfo X] = cat.waitingList.deref[]

def makeSynchWrappers[:X:][pol:PolicyType,cat:CategoryInfo X] =
        record
            preWrapper=MakePreWrapper[pol,
                                    cat.GenMethRef,
                                    cat.ArrivalAction.deref[],
                                    cat.StartAction.deref[],
                                    cat.TermAction.deref[],
                                    cat.Guard.deref[],
                                    cat.waitingList],
            postWrapper=MakePostWrapper[pol]
        end
```

We extended the `CategoryInfo` record with a field holding the waiting list. This allows us to use a global definition for `waitingList` (which is therefore removed from the preprocessor). The other modifications in the code are only used to update this list. Note that we had to apply the same idea as in 4.3.1 to remove an invocation from the list (the field `RemoveYourSelf` in `InvocationType`).

4.4 Removing the preprocessing phase

The careful reader will have noticed that, from the first implementation to the last, the preprocessing phase has been constantly reduced. The ultimate question is then: can we completely remove it? The next subsections will show that the answer is yes.

4.4.1 The new syntax

The current syntax used for a policy definition does not totally correspond to the Pict syntax. But it looks more or less like a function definition, so that we can try to turn it into a real Pict function. We explored more than one possible syntactic form (each having its advantages and disadvantages) and finally chose the following one:

```
abs [Policy,[method-category1,...,method-categoryN]] =
    let
        local-pict-declarations
    in
        Map[Policy,method-category1,
            [ArrivalAction,StartAction,TermAction,Guard]];
        ...
        Map[Policy,method-categoryN,
            [ArrivalAction,StartAction,TermAction,Guard]]
    end
end
```

For example, the policy on page 30 becomes

```
abs [Policy,[Reader,Writer]] =
    let
        def shortestJob[l: List (Invocation Int),len:Int] =
            inFoldList[l,abs[x,res] = res && (x.parameters >= len)
                    end,true]
        def ReaderAllowed[i:SimpleInvocation] = exec[Writer]==0
        def WriterAllowed[i:Invocation Int] =
        exec[Reader]+exec[Writer]==0 &&
            shortestJob[waitingList[Writer],i.parameters]
    in
        Map[Policy,Reader,
            [EmptyAction[], EmptyAction[],
                EmptyAction[], ReaderAllowed]];
        Map[Policy,Writer,
            [EmptyAction[], EmptyAction[],
                EmptyAction[], WriterAllowed]];
    end
```

We also need a new syntactic form for binding a policy to an object. However, we first need to introduce the idea behind the policy's new syntax.

4.4.2 Policies as functions

Consider a policy definition after the preprocessing phase (page 30). We basically do four things:

1. bind the method categories to records of type `CategoryInfo`
2. create the local functions `exec` and `waiting` (remember that now `waitingList` is global)
3. evaluate *local-pict-declarations* and *mapping-declarations*
4. return a record with the synchronization wrappers

The main idea behind the new syntax is to put i) and ii) outside of the policy's definition and to merge iii) and iv). More precisely, we will create one `PolicyType` record plus one `CategoryInfo` record for each method category and pass them as parameters to the policy (which is now a function). The result of the function call will not be a record with synchronization wrappers because we have no possibility to extract the parameters' name to generate the record fields. Actually, this record is only there because we separate the creation of the synchronization wrappers from their binding to the object's methods. But if we bring those two things to-

gether then we do not need any return value. This means that a call to a policy will not only create new synchronization wrappers but will also attach them to the object's methods. Therefore, we need to pass a reference to the wrapped methods as well. This leads us to the following type definitions:

```
type SynchCounterType = Record
                                incr : ![Sig],
                                value : ![!Int]
                            end

type SynchCountersType = [SynchCounterType,SynchCounterType,
                                SynchCounterType]

type InvocationType =   Record
                                Trigger : Sig,
                                ArrivalTime : Int,
                                SynchCounters : SynchCountersType,
                                StartAction : SimpleAction,
                                TermAction : SimpleAction,
                                RemoveYourSelf : SimpleAction,
                                Guard : SimpleGuard
                            end

type PolicyType =       Record
                                Mutex : ^[],
                                Clock : ^Int,
                                WaitingList : Ref (List InvocationType),
                                ExecList : Ref (List InvocationType)
                            end

type SetWrappersType = Fun(X)
                            ![[PreWrapperType X,PostWrapperType],Sig]

type CategoryInfo = Fun (X)
                        Record
                            SynchCounters : [SynchCounterType,
                                            SynchCounterType,
                                            SynchCounterType],
                            waitingList : Ref (List (Invocation X)),
                            SetWrappersList : List
                                                (SetWrappersType X)
                        end
```

The main change here concerns the synchronization counters. They have been removed from the `PolicyType` record and spread over the different `CategoryInfo` records. This was necessary in order to make the functions `exec` and `waiting` global

```
def exec[:X:][cat:CategoryInfo X] =
    let
        val [_,StartCounter,TermCounter] = cat.SynchCounters
    in
        StartCounter.value[]-TermCounter.value[]
    end

def waiting[:X:][cat:CategoryInfo X] =
    let
        val [ArrivalCounter,StartCounter,_] = cat.SynchCounters
```

```
    in
        ArrivalCounter.value[]-StartCounter.value[]
    end
```

Note that `InvocationType` has changed accordingly. We have simply replaced `GenMethRef` with the three synchronization counters of the corresponding method category. `SynchCounterType` represents a reference to a counter that can be incremented by `incr` and read by `value`. The change to the synchronization counters location is also propagated into the code for pre- and post-synchronization wrappers. We will only list the new definition for `MakePostWrapper` (`EvaluateGuards` and `MakePreWrapper` are modified in the same way)

```
def MakePostWrapper[Pol:PolicyType] =
    abs [invr:?InvocationType,r:Sig] >
        invr?Inv >
        Pol.Mutex?[] >
        Pol.ExecList.set[removeInv[Pol.ExecList.deref[],
                                   Inv.ArrivalTime]];
        let
            val [_,_,TermCounter] = Inv.SynchCounters
        in
            TermCounter.incr[]
        end;
        Inv.TermAction[];EvaluateGuards[Pol];Pol.Mutex![]
    end
```

Two other changes have occurred in `CategoryInfo`. First, we removed the references to actions and guards because these are now treated as a block in the `Map` function and thus, do not need to be stored separately any more. Secondly, we added a set of references to methods in the form of a list of functions. Each function allows us to set the synchronization wrappers of one of the methods associated with this method category. They are called in `Map` as follows:

```
def Map[:X:][Pol:PolicyType,cat:CategoryInfo X,
          [ArrivalAction : ActionType X,
          StartAction : ActionType X,
          TermAction : ActionType X,
          Guard: GuardType X]] =
    let
        val Wrappers = [MakePreWrapper[Pol,cat.SynchCounters,
                                       ArrivalAction,
                                       StartAction,
                                       TermAction,
                                       Guard,
                                       cat.waitingList],
                        MakePostWrapper[Pol]]
    in
        inApplyList[cat.SetWrappersList,abs[f] = f[Wrappers] end]
    end
```

`inApplyList` applies a function to every element of a list. `Map` simply creates the synchronization wrappers and binds them to the methods.

4.4.3 Calling a policy

As we have seen in the previous subsection, binding a policy to an object amounts to calling the policy with a `PolicyType` and a tuple of `CategoryInfo` records. The new syntactic form for doing this is:

```
BindPolicy[policy-name,
            [Category[list-of-methods],
                    ...,
            Category[list-of-methods]]]
```

where *`list-of-methods`* is:

```
object.method-name -- extractor @@ ... @@ nil[]
```

Thus, the example on page 28 becomes

```
BindPolicy[ReaderWriterPolicy,
            [Category[o.read--abs [[]] =[] end @@ nil[]],
            Category[o.write--abs [[b,c]] = b end @@ nil[]]]]
```

with `ReaderWriterPolicy` equal to the policy on page 36. The operator `--` returns a function to set the synchronization wrappers of the method

```
def (--)[:X,Y:][mWrapper:MethodWrapper X,
                extractor:ParamExtractorType X Y] =
    abs[[preWrapper:PreWrapperType Y,postWrapper:PostWrapperType]] =
        mWrapper.SetWrappers[preWrapper,postWrapper,extractor]
    end
```

Its purpose is to hide the type `X`, i.e. the type of the method's parameters. The function `Category` creates a `CategoryInfo` record

```
def Category[:X:][l:List (SetWrappersType X)] =
        record
            SynchCounters = [makeSynchCounter[],
                             makeSynchCounter[],
                             makeSynchCounter[]],
            waitingList = ref[nil[:Invocation X:][]],
            SetWrappersList = l
        end
```

while `BindPolicy` calls the policy to wrap the methods

```
def BindPolicy[:X:][PolAbs:PolicyAbs X,par:X] =
            PolAbs[makePolicy[],par]
```

where `PolicyAbs` is

```
type PolicyAbs = Fun(X) ![PolicyType,X,Sig]
```

4.4.4 Variations on the syntax: the *row* operator

It does not seem very natural to have to declare a parameter `Policy` and pass it to the `Map` function. Thus, we would like to write the example on page 36 as follows:

```
abs [Reader,Writer] =
let
    def shortestJob[l: List (Invocation Int),len:Int] =
        inFoldList[l,abs[x,res] = res && (x.parameters >= len)
                    end,true]

    def ReaderAllowed[i:SimpleInvocation] = exec[Writer]==0

    def WriterAllowed[i:Invocation Int] =
        exec[Reader]+exec[Writer]==0 &&
            shortestJob[waitingList[Writer],i.parameters]

in
    Map[Reader,
        [EmptyAction[],EmptyAction[],
        EmptyAction[],ReaderAllowed]];

    Map[Writer,
        [EmptyAction[],EmptyAction[],
        EmptyAction[],WriterAllowed]];
end
```

Actually, it is possible to define a `Map` function in such a way that this syntactic form works. The idea is to extend `CategoryInfo` with a field `policy`, which is a reference to a `PolicyType` record. `Map` is then modified as follows:

```
def Map[:X:][cat:CategoryInfo X,
            [ArrivalAction : ActionType X,
            StartAction : ActionType X,
            TermAction : ActionType X,
            Guard: GuardType X]] =
    let
        val Wrappers = [MakePreWrapper[cat.policy.deref[],
                                        cat.SynchCounters,
                                        ArrivalAction,
                                        StartAction,
                                        TermAction,
                                        Guard,
                                        cat.waitingList],
                        MakePostWrapper[Pol]]
    in
        inApplyList[cat.SetWrappersList,abs[f] = f[Wrappers] end]
    end
```

Now, the problem is to set this field for each method category.

One possibility is to explicitly create the record before calling `BindPolicy` and to pass it as an additional parameter to `Category`:

```
val pol=makePolicy[]
BindPolicy[ReaderWriterPolicy,
            [Category[pol,o.read--abs [[]] =[] end @@ nil[]],
            Category[pol,o.write--abs [[b,c]] = b end @@ nil[]]]]
```

This solution is not very safe because one could use `pol` for two different objects that would then share the same data.

Ideally, we would like to keep the syntax of page 39. This means that we have to scan the tuple of `CategoryInfo` records to initialize their `policy` field. Thus, we would write `BindPolicy` as follows:

```
def BindPolicy[:X:][PolAbs:PolicyAbs X,par:tuple-of-CategoryInfo] =
    let
        pol = makePolicy[]
        def setpol[:X:][cat:CategoryInfo X] =
            cat.policy.set[pol]
    in
        inApplyTuple[par,setpol];
        PolAbs[par]
    end
```

Here *tuple-of-CategoryInfo* is a type representing a tuple of any length of `CategoryInfo` records. *inApplyTuple* is the tuple's counterpart of `inApplyList` introduced on page 38. Such a generic tuple type (and its associated function *inApplyTuple*) does not currently exist in the Pict type system but would be a useful extension. However, its introduction raises some typing issues that first need to be solved, which is beyond the scope of this paper. In the following paragraphs we will just state more clearly what we really understand by "generic tuple type".

This idea of "generic tuple type" comes from LISP where we can define functions with an undefined number of parameters. This features turns out to be very useful when writing generic code and we would like to be able to do the same thing in Pict. For this reason we introduce a new type operator `row` that can only be used inside a tuple type. For example the following type

```
[Int, row String, Sig]
```

stands for any of these types

```
[Int,Sig]
[Int,String,Sig]
[Int,String,String,Sig]
[Int,String,String,String,Sig]
...
```

Thus, at first `row` seems to be a type operator of *kind* `Type -> Type`. But in fact this is not sufficient to express the `BindPolicy` function because `par` stands for a tuple of `CategoryInfo` record of different types (e.g., `CategoryInfo Int` and `CategoryInfo []`). In that case `par` has the type

```
[row CategoryInfo]
```

which means that this time `row` is of kind `(Type -> Type) -> Type`. This can also be seen in the polymorphic function `setpol`. Those two examples show that `row` should be viewed as a *polymorphic type operator* of the form

```
type row = Fun[:K:](X:K) ....
```

where K represents a kind. It is not yet quite clear (at least for the author) how to integrate such an operator in the Pict type system. Actually, other problems occur with this operator. Consider for example a function that takes a tuple of values and returns a tuple of reference cells initialized with those values (*`mapTuple`* is the tuple's counterpart of `mapList`)

```
type Dummy = Fun(X) X
def makeCells[r:row Dummy] : [row Ref] =
    [mapTuple[r,ref]]
```

The type of `makeCells` is

```
![row Dummy,![row Ref]]
```

but this is not sufficient. Actually, we would like to express the fact that `makeCells` returns a tuple with *exactly the same number* of elements as its argument and with the *corresponding* reference types. Given the type above, the result of

```
makeCells[1,'a',true]
```

can be any tuple made of reference cells like

```
[ref[1],ref['a'],ref[true],ref[2]]
```

or

```
[ref['a'],ref[true],ref[1]]
```

This means that with the signature of `makeCells` one cannot typecheck the following program:

```
val [a,b,c] = makeCells[1,'a',true]
val d = a.deref[] + 3
```

although it seems pretty reasonable. To solve this problem, we have to relate the argument to the result in some way, for example by providing a new kind of bound:

```
def makeCells[:X ∈ row Dummy:][r:X] : [Ref X] =
    [inMapTuple[r,ref]]
```

Here `X ∈ row Dummy` means that `X` is an “instance” of type `row Dummy` and `Ref X` is the type we get by applying the `Ref` operator to each element of `X`. There are certainly other related issues that remain to be explored but we will stop here by just giving two type equivalences about the `row` operator

```
[...,row Y,row Y,...] is equivalent to [...,row Y,...]
[...,row (row Y),...] is equivalent to [...,row Y,...]
```

To see why the second equivalence should hold, consider the unfolding of `[...,row (row Y),...]`, that is

```
[...,row Y,row Y,...,row Y,...]
```

which is equivalent to `[...,row Y,...]` by the first equivalence.

5 Conclusion

We have shown an implementation of the “Generic Synchronization Policies” in Pict using a step-by-step approach. The final implementation does not rely on preprocessing any more, which is the goal that we pursued. However, we didn’t really answer the question: Are Pict’s records the right abstraction on which to build an object model ? We just proposed a more convenient syntax (page 11) that should be considered as a syntactic sugar and nothing else. The reason why we did not answer the question is that we did not need to, due to the fact that the GSP

concept is quite independent from the underlying object model. This also means that other experiments are required to bring a satisfying answer. For example, we could imagine implementing a MetaObject Protocol like that of CLOS [3] or the Sina's object model [1]. In those two cases it seems that treating methods' names as first-class values is an essential feature which is not present in current Pict's records. What is also missing is object identity. It could be implemented using channels but then we would need an operator to test the equality of channels.

Another interesting question is to see how the work of Laurent Dami [2] on the lambda-calculus can be applied to the polyadic π-calculus.

6 Acknowledgements

The author would like to thank Benjamin Pierce, Jean-Guy Schneider, Markus Lumpe and Oscar Nierstrasz for commenting and correcting earlier versions of this report.

References

[1] Mehmet Aksit, "On the Design of the Object-Oriented Language Sina," Ph.D. thesis, University of Twente, 1989.

[2] Laurent Dami, "Software Composition: Towards an Integration of Functional and Object-Oriented Approaches," Ph.D. thesis No. 396, University of Geneva, 1994.

[3] Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press (Ed.), 1991.

[4] Ciaran McHale, "Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance," Ph.D. Dissertation, Department of Computer Science, Trinity College, Dublin, 1994.

[5] Robin Milner, Joachim Parrow and David Walker, "A Calculus of Mobile Processes, Parts I and II," Reports ECS-LFCS-89-85 and -86, Computer Science Dept., University of Edinburgh, March 1989.

[6] Benjamin C. Pierce, "PICT: An Experiment in Concurrent Language Design," PICT Version 3.6 tutorial, University of Edinburgh, March, 1994.

[7] Benjamin C. Pierce and David N. Turner, "Concurrent Objects in a Process Calculus," *Proceedings Theory and Practice of Parallel Programming (TPPP 94)*, Springer LNCS 907, Sendai, Japan, 1995, pp. 187-215.

[8] David N. Turner, "The Polymorphic Pi-calculus: Theory and Implementation," Ph.D. thesis, University of Edinburgh, 1995, in preparation.

Implementation of “Generic Synchronization Policies” in Pict

Patrick Varone

IAM-96-005
April 1996

1. Introduction . . . 1
   1.1. Typesetting conventions . . . 1
2. Generic Synchronization Policies . . . 2
   2.1. Definition . . . 2
   2.2. Examples . . . 3
3. Pict . . . 5
   3.1. Channels and types . . . 5
   3.2. Processes . . . 6
   3.3. Local declarations . . . 7
   3.4. Derived forms . . . 7
      3.4.1. The def construct . . . 8
      3.4.2. Functions as processes . . . 8
      3.4.3. Infix operators . . . 9
      3.4.4. Sequencing . . . 9
   3.5. The object model . . . 10
4. Implementation of GSP in Pict . . . 10
   4.1. Synchronization wrappers in Pict . . . 10
      4.1.1. Synchronization wrappers and GSP . . . 10
      4.1.2. Binding a GSP to an object . . . 11
   4.2. GSP in Pict: a first implementation . . . 14
      4.2.1. The shared data structure . . . 14
      4.2.2. Generic code for pre- and post- synchronization functions . . . 15
      4.2.3. Syntax of GSP in Pict . . . 19
   4.3. Accessing the parameters of a method . . . 22
      4.3.1. Storing and retrieving the parameters . . . 23
      4.3.2. Mapping method parameters . . . 28
      4.3.3. Preprocessing . . . 30
      4.3.4. Getting it right . . . 32
   4.4. Removing the preprocessing phase . . . 35
      4.4.1. The new syntax . . . 35
      4.4.2. Policies as functions . . . 36
      4.4.3. Calling a policy . . . 39
      4.4.4. Variations on the syntax: the row operator . . . 39
5. Conclusion . . . 42
6. Acknowledgements . . . 43