ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**D. J. Duke , University of York, UK and A.S. Evans, University of Bradford,
UK (Eds)**

2nd BCS-FACS Northern Formal Methods Workshop

Proceedings of the 2nd BCS-FACS Northern Formal Methods
Workshop, Ilkley, 14-15 July 1997

Formalizing Correlate - from Practice to Pi

B. Robben, F. Piessens and W. Joosen

Springer

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Formalizing CORRELATE - from Practice to Pi

Bert Robben¹, Frank Piessens², Wouter Joosen³

Dept. of Computer Science., K.U.Leuven
Celestijnenlaan 200A, B3001 Leuven - Belgium

Abstract

Correlate is a concurrent object-oriented language that has been used in real-world applications in areas of High-Performance Computing, network applications and multi-agent systems. The practical goal of the language is to offer a powerful programming environment in which both application and system software can be described using the paradigm of concurrent objects.

In this paper, the formal description of Correlate in the pi-calculus is discussed. In our experience, such a formalization process is a crucial step in the design process of a practical language, even though not every detail need to be modelled. Several examples illustrate the benefits of our specification especially from the viewpoint of the language designer. Finally it is shown that an operational prototype can be easily constructed by translation from the pi-calculus to Pict.

1 Introduction

In this paper, we report upon the process of formalising a concurrent object-oriented language, called CORRELATE. The CORRELATE language is a research prototype developed by the distributed systems group of K.U.Leuven. One of the aims of our research team is to develop system software with an open implementation: while applications are developed, system software is being customised to match the application's needs. The practical goal of the CORRELATE language is to offer a powerful programming environment in which both application and system software can be described using the paradigm of concurrent objects. We have gathered experience in using the prototype with applications in the areas of High-Performance Computing (Molecular Dynamics simulation, PDE solvers)[6], Network Applications (routing) and multi-agent systems[7]. This iterative process of experimenting with real-world applications has been driving the continuous improvement of the language.

In this paper, we discuss the formal description of CORRELATE in the π -calculus[9]. In our experience, formalisation is a crucial ingredient in the design process of a practical language, even though not every detail (in this case of CORRELATE) has been (and can be) modelled. We believe that our work is a complement to the formal work that is produced with simple, yet pure, languages where the formalisation is the main goal. In this work, we try to bridge the gap between the pure theoretical work on process calculi, and the development of concurrent object-oriented languages with practical use.

The paper is structured as follows. Section two contains an overview of CORRELATE. Section three briefly talks about the π -calculus. Section four treats the formal semantics of the language. Section five addresses the limitations of the formal model with respect to the full language and evaluates the impact of this limited (yet powerful) model, especially from the viewpoint of a language designer. We discuss related work in section six and conclude in section seven.

2 CORRELATE in a Nutshell

CORRELATE is a statically typed, class-based, object-oriented language. In this sense, it much resembles C++⁴. The focus of the language is on high-level support for developing distributed concurrent object-oriented applications. On the one hand this is realized by a meta-level architecture that gives the application programmer control over the *execution environment* of the application (i.e. things like distribution and persistence). On the other hand and orthogonal to it, new language features are introduced to enable the description of the application as a set of concurrent objects. The remainder of this section will treat the latter topics in more detail.

¹Research assistant of the Belgian National Fund for Scientific Research (N.F.W.O.)

²Postdoctoral fellow of the Belgian National Fund for Scientific Research (N.F.W.O.)

³Research assistant of the Flemish I.W.T.

⁴The CORRELATE compiler is actually a translator to C++.

2.1. Active Objects

A CORRELATE object is a unit of concurrency; only one operation can be executing on an object at the same time. An important concern in concurrent environments is the synchronisation of objects. Depending on the state of the object, a specific operation may or may not be executed. In other words, in a certain state an object can accept only a subset of its entire set of operations in order to maintain its internal integrity. This issue can be expressed with synchronisation constraints. Each operation can be annotated with a precondition that evaluates to true whether the object is ready to accept the operation in its current state or not.

To maintain encapsulation as much as possible and reduce the effects of the inheritance anomaly, CORRELATE objects expose an intermediate abstraction layer which basically corresponds to an *abstract state machine*. At the level of a class definition, the language's syntax provides a *behaviour* section that describes the abstract states as simple functions of the state of the object, i.e. no object interaction or side effects are allowed. Preconditions are then specified as expressions of these abstract states and the parameters of the invocation.

2.2. Autonomy

In the CORRELATE object model an object can exhibit *autonomous behaviour*. This way an object can perform a spontaneous action which is not triggered by external stimuli (incoming messages).

At the level of a class definition, the language's syntax makes a distinction between autonomous operations and interface operations. Interface operations represent the reactive behaviour and are conceptually the same as public operations in object-oriented languages like Java or smalltalk. *Autonomous* operations are invoked as soon as the object has been created and they are re-invoked as soon as the execution of the operation is terminated.

```
active Controller {
  autonomous:
    void Check() {
      if (too_warm) StopHeating();
      if (too_cold) StartHeating();
    }
  interface:
    void TurnOff();
    void RegulateTemperature(Temp tmin, Temp tmax);
  behaviour:
    Boolean IsOn();
    for void Check() precondition IsOn();
  implementation:
    Heater _heater;
};
```

Figure 1: The Controller class

Figure 1 describes a controller for a thermal system. This controller is tied to a heating system and ensures a constant temperature by starting and stopping a heater.

2.3. Object Interaction

Object interaction happens by sending messages (invoking interface operations). Different models for message passing exist. Synchronous message passing blocks the sender's activity until the operation is executed to completion in the receiver object. With asynchronous message passing, the sender does not have to wait for the completion of the invocation, thus increasing concurrency. CORRELATE supports both *synchronous* and *asynchronous* message passing. The application programmer must explicitly specify⁵ what kind of message passing (s)he prefers.

In order to express a message based interaction, the sender must specify the recipient of a particular message. Most programming models employ a name-based scheme to indicate the recipient. In addition to this, CORRELATE supports a more powerful scheme in the form of pattern-based group communication[3]. Pattern-based group communication is a facility to specify sender-initiated coordination towards a group of recipients based on some of their properties.

⁵A synchronous invocation is annotated with a '\$' character, an asynchronous one with '@'.

```

active Ship {
autonomous:
  void Sail() {
    MoveInCurrentDirection();
    if ( inTrouble )
      Ship[InRange(myCoordinates)] @ SOS(Self);
  }
interface:
  void SOS(Ship shipInTrouble);
  ...
behaviour:
  bool InRange(Coordinate pos);
  ...
};

```

Figure 2: Implicit Interaction

All objects of the specified class that match the condition⁶ participate in the communication and are sent a message. In the example shown above, when a ship gets into trouble, it will send an SOS message to all ships that are near. *Self* is an implicit variable referring to the object itself (like ‘this’ in Java/C++ and ‘self’ in Smalltalk).

3 A Short Introduction to the π -calculus

The π -calculus is an elementary calculus for describing and analyzing concurrent systems. It is not possible to explain in detail all its aspects in a short space. Therefore, this overview is only meant as a quick refreshing of memory. Interested readers not familiar with the π -calculus can find an indepth discussion in [9].

A system is defined as a collection of independent processes that communicate through named channels. Processes are built as follows:

- $P \mid Q$ will execute both P and Q *concurrently*.
- The replication operator $!$ provides an infinite number of concurrent copies of a given process. $!P = P \mid !P = P \mid P \mid P \mid \dots$. It can be shown that recursion can be encoded as replication, so it does not have to be tackled separately.
- $(\nu x) P$ introduces a new name x with scope P . All free occurrences of x in P are bound to the new name.
- Processes can communicate with each other over channels.
 $\bar{x}(z).P$ outputs z over the channel x and then executes P .
 $x(y).P$ receives a name along x and then executes P with each free occurrence of y in P substituted by this received name.
 Communication happens synchronously: a name can only be sent when a corresponding receiver process is ready to accept it.
- $P + Q$ will execute *either* P or Q .
- 0 stops. It is the identity process for $\mid$ and $+$.

As a simple example, consider the process $\bar{x}(y) \mid x(z).\bar{z}(v) \mid \bar{x}(w)$. One of two communications (but not both) can occur along channel x ; either name y or name w will be sent. The two alternatives for the result are $0 \mid \bar{y}(v) \mid \bar{x}(w)$ and $\bar{x}(y) \mid \bar{w}(v) \mid 0$. The (indeterministic) result of the communication thus determines over which channel (either y or w) name v will be sent.

Milner’s tutorial[9] further describes a set of formal reduction rules through which processes can be reduced. The calculus as presented up until now is *monadic* - i.e. only one name is sent over a channel at the same time. A *polyadic* form where multiple names can be sent at once over a single channel is much more convenient in many cases; it can be shown that this is no real extension to the calculus as it can be encoded in the monadic form. Milner further introduces *sorts*, which is a kind of type-system for channels in this polyadic form. The idea is that e.g. given a channel x has sort $(A \ B \ C)$, all communication along x involves three names (say a , b and c) with a corresponding sort of A , B and C . This way, well-sorted processes never arrive in a situation where the sender sends a different number of names from what the receiver expects. A last extension is to allow not only names, but also full-fledged

⁶Like a precondition, this condition is a boolean function of its parameters and the abstract states.

processes to be sent along a channel. It can be shown that this *higher-order* π -calculus can be encoded in the basic (*first-order*) π -calculus. The formal specification of CORRELATE will be presented by translation to the first-order polyadic sorted π -calculus.

4 Process Calculus Semantics

This section gives the formal semantics of the CORRELATE language. In the first subsection, the basic data types are encoded. Subsection two describes the internal view on an object. This serves as the basic architecture on top of which the various language statements and expressions can be built in subsection three.

To enhance readability, the various entities are annotated in different styles: sorts are annotated in SMALL CAPS, names of processes in normal font and channels in *italics*.

4.1. Basic Data Types

The Correlate language supports four different kinds of values: active objects, booleans, integers and strings. Each of these entities will be represented by a process with a single free variable serving as (initial) communication channel. The boolean value true and the integer value 5 can e.g. be encoded as :

```
True(b) = ! b(t f). $\bar{t}$ 
Five(i) = ! i(s z). $\bar{s}.\bar{s}.\bar{s}.\bar{s}.\bar{z}$ 
```

Strings can be represented in a similar way. The exact coding of these basic data types is however not essential to this work, so the point is not further discussed. Possible encodings can be found in [9]. The representation of active objects is treated in detail in the next subsection.

Given that all values can be represented as a single channel, it seems straightforward to model variables as locations that contain values (see also [9]). Simple Get and Set processes can be defined to respectively read and overwrite the contents of a given variable.

```
Create(lCh v) = (v loc) (  $\overline{loc}(v)$  |  $\overline{lCh}(loc)$  )
Get(loc res) = loc(v).( $\overline{loc}(v)$  |  $\overline{res}(v)$  )
Set(loc newV done) = loc(v).( $\overline{loc}(newV)$  |  $\overline{done}$  )
```

This solution however is not completely satisfying since the sort of each variable depends on its type. Indeed, as shown further on, an active object will not be represented like integers and booleans as a channel of sort $((\)\)$. This difference in sorts makes it difficult to write generic well-sorted processes that work for any variable regardless of its type (like for instance an assignment process). A simple extra indirection however is enough to restore well-sortedness.

```
VALUE -> ((OBJECTREFERENCE) (BOOLEAN) (INTEGER) (STRING))
LOCATION -> ( VALUE )
```

A value is now implemented as a process that expects four channels (one for each basic type) as input. It will output its data over the channel corresponding to its type. Integer and boolean values are thus represented as follows.

```
IntVal(val i) = ! val(och bch ich sch). $\overline{lCh}(i)$ 
BoolVal(val b) = ! val(och bch ich sch). $\overline{bCh}(b)$ 
```

These processes form the basic structure to represent variables in the language semantics. Before the structure of active objects is tackled, two “sugar processes” are introduced that are used further on and ease the use of VALUE processes. A first simple process is TrueVal; it checks whether a given value is of type Bool and signals channel *t* if it is true and *f* otherwise.

```
TrueVal(val t f) = (v och bch ich sch) (
   $\overline{val}(och\ bch\ ich\ sch).bch(b).\bar{b}(t\ f)$ 
)
```

A second useful process that is frequently used further on is an aggregation of locations. Such a process can be used to model a stack or a list of variables. Again, the exact representation of these LocationLists is not essential, so only a graphical form is presented here. Appropriate processes are defined to create a list (EmptyList), to add an element (Cons) and to iterate over all elements (ForEach). A specific element in a list can be referred to by a Slot and

directly addressed (Lookup).

Figure 3: Locations and LocationLists

4.2. Objects

This section describes a basic runtime architecture to support concurrent objects. After a global overview, a detailed encoding of the different processes is presented.

4.2.1 The Big Picture

An object is a composition of state (membervariables) and behaviour (i.e. how an object acts and reacts). An active object requires in addition a synchronisation mechanism to deal with concurrent requests. One Object process is created for each object to encapsulate all this information.

Operations are modelled as OperationExecutor processes. As Correlate is a class-based language, all objects of the same class share the same operations. In other words, for each operation defined on a class one OperationExecutor process is running. It is their responsibility to enforce the synchronize constraints to ensure the correct Correlate semantics. The Lock associated with each object is used to implement this.

Correlate objects can refer to each other. However, object interaction is *only* possible through message passing; the internal state of an object is only accessible to the object itself. This is directly presented in the semantics through the ObjectReference processes. Such a process is directly linked to a single Object process, but only exports a mechanism to invoke operations (as opposed to Object processes that allow e.g. direct manipulation of the object's internal state). ObjectReferences are considered basic values and then can as such be stored in locations.

Objects are created by ObjectCreator processes. One such process exists for each class. Destruction of an object is done through a DestructorExecutor which is a special kind of OperationExecutor process. Finally, pattern based group communication will be orchestrated by a ClassCoordinator process. Figure 4 gives a graphical overview.

4.2.2 Using Objects

An object is directly represented as a composition of synchronization (a lock), state (a list of membervariables) and behaviour (a list of all interface operations, preconditions, state abstractions and coordination conditions defined on the object).

```
OBJECT -> ( LOCK LOCATIONLIST EXECUTABLEOPERATIONLIST )
Object(o lock mList opList) = !  $\overline{o}$ (lock mList opList)
```

An executable operation process needs three items of information: the object whose operation is to be executed, a list of the actual parameters of the operation and a channel over which the result can be returned. This is expressed in the EXECUTABLEOPERATION sort.

```
RESULT -> ( VALUE )
EXECUTABLEOPERATION -> ( OBJECT LOCATIONLIST RESULT )
```

Appropriate processes can now be introduced to ease the use of objects. They basically query the object for the corresponding resources and then start their respective actions.

```
LockObject(object,done) =
  object(lock memberList exOpList).lock. $\overline{done}$ 
UnlockObject(object) =
  object(lock memberList exOpList).lock
```

Figure 4: Structure of an Object

```

GetMember(object, slot, res) =
  object(lock memberList exOpLst).(v resL) (
    Lookup(memberList, slot, resL) |
    resL(location).Get(location, res)
  )
SetMember(object, slot, value, done) =
  object(lock memberList exOpLst).(v resL) (
    Lookup(memberList, slot, resL) |
    resL(location).Set(location, value, done)
  )
Invoke(object, slot, paramList, res) =
  object(lock memberList exOpLst).(v resExOper) (
    Lookup(exOpLst, slot, resExOper) |
    resExOper(execOper).execOper(object paramList res)
  )

```

Slots are used in the Invoke process to indicate the operation. This indirection between call and execution is the basic execution time mechanism that is required to support inheritance (late binding). As inheritance is however much more an issue of compilation than execution, it is not treated further here, but in view of future work, this basic functionality is already set up.

4.2.3 Object References

An ObjectReference process can now easily be represented through the following process.

```

OBJECTREFERENCE -> ( SLOT LOCATIONLIST RESULT )
ObjectReference(oRef, object) =
  ! oRef(slot pList res).Invoke(object, slot, pList, res)

```

Notice that two different ObjectReference processes can refer to the same object as the identity of an object is captured by the channel over which its basic components are sent (the *o*-channel in the Object process definition) and not by the ObjectReference channel.

4.2.4 Executing Operations

As Correlate doesn't allow intra-object concurrency, a synchronization check is required before execution of an operation on an object. This is implemented through a simple locking mechanism that ensures that all operations on an object run inside a critical section. User-defined synchronisation constraints in the form of precondition can be easily supported: inside the critical section, but before executing the method, the precondition is computed. If this evaluates to true, the operation can start. Otherwise, the critical section is ended (i.e. the lock is released) and the operation is reapplied. The following *OperationExecutor* process implements this.

```

OperationExecutor(execOperation,preConSlot,execStat) =
  ! execOperation(object locationList return).(v done)(
    LockObject(object,done) |
    done.(v res t f) (
      Invoke(object,preConSlot,locationList,res) |
      res(v).TrueVal(v,t,f) |
      t.(v d) (
        execStat(object locationList res d) |
        res(v).( UnlockObject(object) | return(v) )
      ) |
      f.(
        UnlockObject(object) |
        execOperation(object locationList return)
      )
    )
  )

```

Notice that no scheduling decisions are specified at all; when two operations are competing for access to an object, neither has an advantage over the other.

Given such an *OperationExecutor*, autonomous operations can be modelled quite easily as operations (without parameters) that are reinvoked continuously.

```

AutoOperationCaller(execOp,object) = (v res emptyCh) (
  EmptyList(emptyCh) |
  emptyCh(paramList).execOp(object paramList res) |
  res(v).AutoOperationCaller(execOp,object)
)

```

For each interface or autonomous operation, one *OperationExecutor* process will be active at run-time. In contrast, each *object* will have its own set of *AutoOperationCaller* processes running.

4.2.5 Creation and Destruction

Creating an object is a somewhat complicated matter. First, a new name for the object is created and new locations are allocated for each member variable. Then, the constructor is executed to completion. Afterwards, the autonomous operations can start. Finally, the existence of the new object is communicated to the *ClassCoordinator*.

```

ObjectCreator(execNew,cc,autoLst,exOpLst,memberLst) =
  ! execNew(slot paramList result).
    (v oRef object lock locLstCh res value) (
      MemberInstantiator(memberLst,locLstCh) |
      locLstCh(locLst).! object(lock locLst exOpLst) |
      Lock |
      Invoke(object,slot,paramList,res) |
      res(v).(v done) (
        AutoStarter(autoLst,object) |
        InformCoordinator(cc,oRef,done) |
        done.ObjectReference(oRef,object)
      ) |
      result(value) | ObjectVal(value,oRef)
    )

```

In this code fragment, the *execNew* channel expects the slot corresponding to the constructor operation, a list of actual parameters for this constructor and a result channel over which a reference to the newly created object can

be returned. The actual constructor itself is represented by an `OperationExecutor` process. Each class will have one `ObjectCreator` process active at run-time. The `AutoStarter` processes iterates over a list of all autonomous operations and creates for each of them an `AutoOperationCaller` process.

```

AutoStarter(autoList,object) =
  (v iterator done locListCh) (
    ForEach(autoList,iterator,done) |
    ! iterator(autoOperation,d).(
      AutoOperationCaller(autoOperation,object) |  $\bar{a}$ 
    )
  )

```

One way of destroying an object would be to halt the communication of the object's resources by the `Object` process. This approach is not followed here. Instead, an object is destroyed by never leaving its critical section. This is equivalent to destruction because no operation is allowed to execute when the critical section is not left. The following `DestructorExecutor` process implements this. Notice the similarity with the `OperationExecutor` process.

```

DestructorExecutor(execOperation,preConSlot,execStat) =
  ! execOperation(object locationList return).(v done)(
    LockObject(object,done) |
    done.(v res t f value) (
      Invoke(object,preConSlot,locationList,res) |
      res(v).TrueVal(v,t,f) |
      t.(v r d)  $\bar{execStat}$ (object locationList r d) |
      f.(v r) (
        UnlockObject(object) |
         $\bar{execOper}$ (object locationList r)
      ) |
       $\bar{return}$ (value)
    )
  )

```

One `DestructorExecutor` process is active for each destructor (i.e. for each class).

4.2.6 Coordination Support

Implicit invocations are handled by a `ClassCoordinator`. For each class, one such process is created. It accepts implicit invocation requests, evaluates all objects of its class concurrently to check whether they meet the specified condition and executes the requested operation on those meeting it.

```

ClassCoordinator(coordinator,objectList) =
  ! coordinator(condSlot condParams opSlot opParams).
  (v iterator done) (
    ForEach(objectList,iterator,done) |
    ! iterator(objRef d).(v res t f) (
       $\bar{a}$  |
       $\bar{objRef}$ (condSlot condParams res).res(v).
      TrueVal(v,t,f) |
      t. $\bar{objRef}$ (opSlot opParams res)
    )
  )

```

As the `ClassCoordinator` is notified each time a new object is created, it can create a simple list to keep track of all objects of the class and update it at each object creation.

```

InformCoordinator(objectList,objRef,done) = Cons(objectList,oRef,done)

```

4.3. Statements and Expressions

This subsection explains the internal details of the `OperationExecutor` processes. In other words, it will explain the translation of the different statements and expressions in the CORRELATE language and show how they are linked together to form operations. For clarity, each process that directly corresponds to a Correlate language entity will be preceded by the Correlate statement/expression it specifies between square brackets.

4.3.1 The Big Picture

Essentially, the structure is very simple. The code of an operation is transformed into an abstract syntax tree. Each node in the tree, i.e. each language entity, will be represented by its own process. The example in Figure 5 features 7 processes. The corresponding executable operation will be tied to the top-level node, in this case the Sequence process.

Figure 5: A Simple Statement

A closer look reveals that the different language entities can be partitioned into two groups: statements and expressions. An expression needs to know on which object it handles (e.g. in case it references member variables) and in which environment it is executed (e.g. to bind local variables). The outcome is always a value.

EXECEXPRESSION $\rightarrow$ (OBJECT LOCATIONLIST RESULT)

Figure 6: Expressions

A statement is similar to an expression but it can have two possible outcomes. It can either terminate or return a value. The latter case is used to model the “return” language statement, which indicates that the top-level operation is finished. The other statements (like e.g. assignment to a variable) just terminate meaning that the next statement in the sequence can be executed. This difference is apparent in a Statement process in that it has an additional parameter (the *done* channel) with respect to expressions.

EXECSTATEMENT $\rightarrow$ (OBJECT LOCATIONLIST RESULT ())

Figure 7: Statements

4.3.2 Flow of Control

The processes that define flow of control constrain how different statement can be combined into an executable operation. Sequential execution defines that a statement can only be executed when the previous statement has completed. Notice that if the first statement never finishes (as is the case e.g. for a “return” statement), the second

statement is not executed. Conditional execution is provided by an if-then-else process, iteration by the while process. The return statement provides a mechanism to end the execution of the operation.

```
[ S1;S2 ]
SeqExecutor(execStat,execS1,execS2) =
  ! execStat(object,locList,result,done).(v d) (
     $\overline{\text{execS1}}$ (object,locList,result,d).d. $\overline{\text{execS2}}$ (object,locList,result,done)
  )
[ return E ]
ReturnExecutor(execS,execE) = ! execS(obj locList res d). $\overline{\text{execE}}$ (obj locList res)
[ if Expr then S1 else S2 ]
IfThenElseExecutor(eStat execExpr execS1 execS2) =
  ! eStat(object locList result done).(v res t f) (
     $\overline{\text{execExpr}}$ (object locList res) |
    res(v).TrueVal(v,t,f) |
    t. $\overline{\text{execS1}}$ (object locList result done) |
    f. $\overline{\text{execS2}}$ (object locList result done)
  )
[ while Expr do S ]
WhileExecutor(execStat,execExpr,execS) =
  ! execStat(object locList result done).(v res t) (
     $\overline{\text{execExpr}}$ (object locList res) |
    res(v).TrueVal(v,t,done) |
    t.(v d) (
       $\overline{\text{execS}}$ (object locList result d) |
      d. $\overline{\text{execStat}}$ (object locList result done)
    )
  )
```

4.3.3 Working with Objects

Through *Self* an object can refer to itself. The corresponding process simply creates a new ObjectReference process referring to the current object.

```
[ Self ]
SelfExecutor(execExpr) =
  ! execExpr(object locList result).(v objectRef) (
    ObjectReference(objRef,object) |
    (v value) ( ObjectVal(value,objRef) |  $\overline{\text{result}}$ (value))
  )
```

The *New* operator creates a new object. It first computes all constructor arguments and then contacts (through the *execNew* channel) the ObjectCreator corresponding to the respective class to actually create the object. From the definition of this process, it follows that the new operator completes when the constructor is started. The process is however carefully coded to ensure that no operations are actually started before the constructor is completed.

```
[ New ClassName(e1,...en) ]
NewExecutor(execExpr,execNew,slot,paramList) =
  ! execExpr(object locList result).(v locListCh res) (
    ExecuteFLTR(paramList,object,locList,locListCh) |
    locListCh(actParams). $\overline{\text{execNew}}$ (cons actParams result)
  )
ExecuteFLTR(exExprList,object,locList,locListCh) = (v iterator done elocListCh) (
  EmptyList(elocListCh) |
  ForEach(exExprList,iterator,done) |
  elocListCh(paramList).(
    ! iterator(exExpr d).(v res loc) (
       $\overline{\text{exExpr}}$ (object locList res).res(val).(  $\overline{\text{loc}}$ (val) | Cons(paramList,loc,d) )
    ) |
    done. $\overline{\text{locListCh}}$ (paramList)
  )
)
```

The *Delete* operator destroys the object computed by the expression. Notice that there is no synchronisation with the actual destruction of the object ! The caller has no idea when the victim is actually deleted.

```
[ Delete Expr ]
DeleteExecutor(execStat,destSlot,execExpr) =
  ! execStat(obj,locList,result,done).(v res) (
    execExpr(obj locList res) |
    res(v).(v eLstCh) (
      EmptyList(eLstCh) |
      eLstCh(eList).Call(v,destSlot,eList,res) |
      done
    )
  )

Call(value,slot,paramList,res) = (v oCh bCh nCh sCh)(
  value(oCh bCh nCh sCh).oCh(objectRef).
  objectRef(slot paramList res)
)
```

4.3.4 Object Interaction

Synchronous and asynchronous invocations are much alike. In both cases, first all actual arguments are computed (from left to right), then the receiver object is computed and finally the call is made. Only from that point on do they differ; an asynchronous invocation signals the *done* channel concurrently with executing the call, while the synchronous call waits for the reply.

```
[ expr $ M(e1,...en) ]
SynchCallExecutor(execStat,exExpr,mSlot,exExprList) =
  ! execStat(object locList result done).(v resList) (
    ExecFLTR(exExprList,object,locList,resList) |
    resList(mParams).(v res) (
      execExpr(object locList res).res(rcvr).(
        Call(rcvr,mSlot,mParams,res) | res(v).done
      )
    )
  )
```

When a synchronous call functions as an expression, the result will not be thrown away like above (where it is a statement), but it will be returned as the result of the expression (i.e. over the *result* channel).

```
[ expr @ M(e1,...en) ]
AsynchCallExecutor(execStat,exExpr,mSlot,exExprList) =
  ! execStat(object locList result done).(v resList) (
    ExecFLTR(exExprList,object,locList,resList) |
    resList(mParams).(v res) (
      execExpr(object locList res).res(rcvr).(
        Call(rcvr,mSlot,mParams,res) | done
      )
    )
  )
```

In case of an implicit invocation, both the arguments of the condition (*cParams*) as those of the operation (*mParams*) are computed. When the class coordinator is finally contacted, the execution of the statement is finished.

```
[ Class[C(e1',...en')] $ M(e1,...en) ]
ImplicitCallExecutor(execStat,coordinator,cSlot,cExprLst,mSlot,mExprLst) =
  ! execStat(object,locList,result,done).(v resList)(
    ExecuteFLTR(cExprLst,object,locList,resList) |
    resList(cParams).(
      ExecuteFLTR(mExprLst,object,locList,resList) |
      resList(mParams).coordinator(cSlot cParams mSlot mParams).done
    )
  )
```

4.3.5 Variables

Declaration of a new local variable results in creation of a new location which is added to the current stack. The location is initialized with a compiler-selected value depending on the type of the variable.

```
[ Type VarName ]
VarDeclExecutor(execS,value) =
  ! execS(object locList result done).(v locCh) (
    Create(locCh,value) |
    locCh(location).Cons(locList,location,done)
  )
```

Lookup and assignment depend on whether the variable is local or a member variable, but otherwise follow much the same pattern.

```
[ VarName ]
VariableExecutor(execExpr,slot) =
  ! execExpr(object locList result).(v locCh) (
    LookupList(locList,slot,locCh)
    locCh(loc).Get(loc,result)
  )

[ VarName = Expr ]
AssignmentExecutor(execStat,slot,execExpr) =
  ! execStat(object locList result done).(v res locCh) (
    execExpr(object locList resExpr) |
    Lookup(locList,slot,locCh) |
    res(v).locCh(location).Set(location,v,done)
  )

[ MemberName ]
MemberExecutor(execExpr,slot) =
  ! execExpr(obj locList res).GetMember(obj,slot,res)

[ MemberName = Expr ]
AssignMemberExecutor(execStat,slot,execExpr) =
  ! execStat(object locList result done).(v res)
    execExpr(object locList resExpr) |
    resExpr(value).SetMember(object,slot,value,done)
  )
```

4.3.6 Constants and built-in Functions

The most simple expression is that of a constant.

```
[ ConstantValue ]
ConstantExecutor(execExpr,value) =
  ! execExpr(object locList result).result(value)
```

A function reflects a built-in operation on one of the basic data types. Examples are equality operators, arithmetic operations and string concatenation. The result is only dependent on the parameters and produces no side-effects.

```
[ FunctionName(e1,...en) ]
FunctionExecutor(execExpr,execFunc,execExprList) =
  ! execExpr(object locList result).(v lCh) (
    ExecuteFLTR(execExprList,object,locList,lCh) |
    lCh(paramList).execFunc(paramList result)
  )
```

5 Discussion

The formal semantics given in section four does not cover all aspects of CORRELATE[12]. Very little attention has

been given to typing and inheritance. The meta-level architecture -- together with aspects like distribution and persistence -- was not touched at all. Concurrency and coordination on the other hand were treated in detail. The first subsection shows that although incomplete, the semantics are a very valuable tool both for the language designer and the language run-time implementor. The section concludes with a glance on some future work.

5.1. Benefits of the Current Model

Expressing the language semantics in a formal and unambiguous way, is very useful to the language designer. Through the specifications, (s)he can clearly state which properties are part of the semantics and which are only side effects of a particular implementation. In CORRELATE for example, the semantics of asynchronous invocations state clearly that no ordering on message acceptance is guaranteed. The current implementation however delivers all messages in the same order as they were sent, but only because of a set of implementation decisions concerning scheduling and network communication which can and most probably will change in a future implementation.

As the π -calculus allows very fine grained concurrency, a clear grasp is obtained on the synchronization possibilities of the language while writing the specification. E.g. in our case, synchronization of interface operations through preconditions was already well studied before creating the π description. Object construction and destruction however were given only ad hoc semantics. In the formal description, the lack of synchronization on e.g. the destructor shows that it is very awkward to write a class where an action should only be taken when a certain object is really destroyed. Hence the formal specification of the language influences the language design itself.

The language run-time implementor also gains from a clear specification. The list of Executor processes gives a complete overview of the basic functionality that is required for executing a CORRELATE program. The development of a new execution environment for the language can now much sooner in the design process be validated for completeness and correctness.

Since the π -calculus has an operational semantics, the specifications have the advantage that they can be executed by an appropriate execution engine. However, the π -calculus is a very abstract formal system and although such execution engines exist[13], extracting meaningful information is not always easy as the gap with a high-level notation suitable for general-purpose concurrent programming is considerable. A lot of work has been done however to close this gap. Pict[10] is a formal concurrent language related to the π -calculus, as LISP is to the λ -calculus. A compiler and execution environment is available. It turned out that our π -calculus semantics could be implemented in Pict with relatively little effort. Some small patches were needed to cope with replicated output (which is not supported in Pict). The biggest difficulty was the difference between sorts and types, and more specifically the handling of (mutually) recursive types as they are only minimally supported (Pict v3.8e). The benefits of the Pict implementation of the formal semantics were twofold: first, it served as a fine test bed for the specification. Both the type checker at compile-time and executing simple programs at run-time are great tools to find hidden faults or confirm a correct specification. A second benefit was that the Pict program is a valid prototype execution environment. As such it can be more easily used to test some new language features than the actual execution environment which is a much bigger program (in C++).

5.2. Future Work

The work presented here is not complete. The specification as is can be extended to address additional functionality. For instance, CORRELATE (like Java) allows code to be dynamically added to a running application. To reason about this formally, a specification of the compilation process must exist. At the moment, such a specification is nearly finished.

A more long term view is to use the specification to prove properties of the architecture and of certain objects. Object encapsulation can serve as a simple example. It should not be too hard to prove, that given only access to an ObjectReference, a process cannot directly access the object's internals. More ambitious, through the concept of unique bearing of names[9] it might be possible to prove that no spoofing (i.e. impersonating another object) will happen during execution. This is particularly important for a language that allows dynamic loading of code from unreliable sources. When loading the code, one wants to make sure that the code will not try to impersonate certain system-objects. In the same sense, if limited use of resources (i.e. whether or not a process uses another process) can be proved of a foreign process, it can be executed safely. For instance, when loading code from an untrustworthy source, one might want to verify that this code does not access certain system-objects. Many classical properties of concurrent systems (like e.g. deadlock free-ness) are also open for investigation.

The notion of (bi)similarity is another opportunity for further study. Through this concept it might be possible to construct a kind of "behavioural" type where not only the static structure of a class determines its type but also

its dynamic interaction pattern. This could then be used to formulate constraints on certain objects by requiring them to be of a specific type. A good example would be the case of meta-objects. Without going in too much detail, a meta-object can be defined as an object that can change the semantics of the language. By allowing the user to write his/her own meta-objects he/she can adapt the language to his/her needs. A problem with meta-objects is that sometimes they can be too powerful. Requiring meta-objects to be of a certain type could be a way of controlling their expressiveness. The use of verification tools like [13] is essential to get practical use, but whether current tools are powerful enough to deal with these complex systems is not clear at the moment.

6 Related Work

The pioneering work on semantics of parallel object-oriented languages was done by America et al. [1,2] on the POOL family of languages. Both an operational and a denotational semantics of (one dialect) of the POOL family of languages is given in [2]. There does not seem to be a general agreement yet, about which formal system should be used as a foundation of concurrent object-oriented languages. Popular formalisms include: various kinds of object-calculi, various fragments or extensions of linear logic, and classical process algebras like CCS, Lotos or CSP. However, it seems to us that, for the kind of properties of CORRELATE that we want to investigate formally (see subsection 5.2.), a π -calculus semantics is a good approach. Indeed, in his tutorial paper [9], Milner argues that the π -calculus is a very appropriate formalism to study these kinds of properties (he gives as examples ‘unique bearing of names’ and ‘unique handling of names’ which are close to the kind of properties we want to investigate).

π -calculus semantics of parallel object-oriented languages have been studied by Walker [14,15] and Jones [5]. Pierce and Turner [11] have also presented a number of possible representations of active objects in PICT, and hence in the π -calculus. The main difference with our work is that we have started from an existing ‘engineering’ language, already used in many real-life projects [6,7]. Other work [4], also presents an existing language, but it is much simpler and closer to the π -calculus, as the core of the language consists of processes and names.

7 Conclusion

This paper provides a formal semantics of the CORRELATE language in the π -calculus. CORRELATE is a practical language used to describe applications and system software in a distributed concurrent environment. The semantics does not specify every detail of the language; yet it has proven to be a valuable tool for both the language designer and the language implementor. The close relationship between the π -calculus and Pict allowed us to construct a running prototype from the specification.

The presented semantics opens further avenues of investigation. On the one hand, the model can be extended to broaden the scope of the specification. In that area, we have been looking at extending the specification to encompass dynamic loading of code. A more ambitious goal is to prove properties of the architecture and of processes. This might be used to guarantee safe execution of code from an unreliable source.

Acknowledgements

We would like to thank Prof. Benjamin Pierce and Dr. Dave Turner for their quick and helpful assistance every time we encountered a problem with the PICT compiler.

References

1. P. America, J. de Bakker, J. Kok and J. Rutten, Denotational Semantics of a Parallel Object-Oriented Language, *Information and Computation*, 83, 152-205 (1989).
2. P. America, Issues in the Design of a Parallel Object-Oriented Language, *Formal Aspects of Computing*, 1, 366-411 (1989).
3. Stijn Bijns, Wouter Joosen and Pierre Verbaeten. Language Constructs for Coordination in an Agent Space. 6th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Proc. MAAMAW '94, Odense Denmark, LNAI-1069. Springer-Verlag 1996.

4. S. Eisenbach and R. Patterson. π -Calculus Semantics for the Concurrent Configuration Language Darwin. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume 2. IEEE Computer Society Press, 1993.
5. C. B. Jones. Process-Algebraic Foundations for an Object-Based Design Notation. Technical Report Dept. of Computer Science, Manchester University, October 1993.
6. Wouter Joosen, Stijn Bijmens, Johan Van Oeyen, Bert Robben, Frank Matthijs, Pierre Verbaeten and John Peram. *Affordable Overhead in CORRELATE: Comparing two MD Simulators*. Proc. High Performance Computing and Networking, LNCS-1067, pages 217-227. Springer-Verlag, 1996.
7. Wouter Joosen, Frank Matthijs, Johan Van Oeyen, Bert Robben, Stijn Bijmens and Pierre Verbaeten. CORRELATE: High-level Support for Travelling Agents. Technical Report CW236, dept. of Computer Science, K.U.Leuven, October 1996.
8. Robin Milner. *Communication and concurrency*. Prentice-Hall international series in computer science, Prentice-Hall London, 1989.
9. Robin Milner. The Polyadic π -Calculus: a tutorial. *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.
10. Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report in preparation; available electronically, 1996.
11. Benjamin C. Pierce and David N. Turner. Concurrent Objects in a Process Calculus, *Theory and Practice of Parallel Programming (TPPP)*, Sendai, Japan Nov. 1994, editor Takayasu Ito and Akinori Yonezawa, Springer-Verlag, LNCS 907, 1995.
12. Johan Van Oeyen, Stijn Bijmens, Wouter Joosen, Bert Robben, Frank Matthijs and Pierre Verbaeten. "A Flexible Object Support System as Run-time for Concurrent Object-Oriented Languages". In Chris Zimmermann, editor, *Metaobject Protocols*, chapter 12, CRC Inc, May 1996.
13. B. Victor. *A Verification Tool for the Polyadic π -Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.
14. David Walker. Process Calculus and Parallel Object-Oriented Programming Languages. In P. Turdik, T. Casavant and F. Plasil, editors, *Parallel Computers: Theory and Practice*, pages 369-390. Computer Society Press, 1993. Available as Research Report CS-RR-242, Dept. of Comp. Science, University of Warwick.
15. David Walker. π -Calculus Semantics for Object-Oriented Programming Languages, in *Proc. TACS'91*, Springer-Verlag LNCS 526, 532-547 (1991).