Implementing POOSL in C++

van Tetrode, J.N.

Award date:
1997

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Section of Information and Communication Systems

Master's Thesis:

Implementing POOSL in C++

By J.N. van Tetrode

Coaches : ir. M.C.W. Geilen
dr.ir. J.P.M. Voeten
dr.ing. P.H.A. van der Putten
Supervisor : prof.ir. M.P.J. Stevens
Period : September 1996 – August 1997

Abstract

The method Software/Hardware Engineering (SHE) offers a formal language called POOSL (Parallel Object-Oriented Specification Language). This thesis describes a method for implementing POOSL in C++. The method incorporates a set of translation-rules and a C++ POOSL library that contains the required functionality to implement POOSL in C++.

Implementation of the data part of POOSL requires a garbage collector for destroying data objects that are not needed anymore. This is because POOSL's *new* statement that is used to dynamically create data objects has no counterpart for object deletion. For this reason, the POOSL library is supplied with a garbage collector. The applied garbage collection technique is reference counting.

Implementation of POOSL's process part is not straightforward. This is because of the synchronous inter-process communication in combination with the select, abort and interrupt statements. A process can not decide for itself with which process it is able to communicate, since this depends on the decisions of possible communication partner processes. To solve this problem efficiently, a scheduler is used for arbitration. The interrupt and abort statements allow several statements to be active simultaneously, each within its own (local) environment. Whether an active communication statement is executable or not, depends on the communication partners. The scheduler's task is to choose environments that have an executable statement and to give these environments permission to execute it. Therefore, prior to executing its active statement, an environment must submit a request and wait for the scheduler to grant it.

The implementation method presented in this thesis supports all POOSL statements including the delay and broadcast extensions. By combining this work with another master's project, namely the construction of a POOSL compiler, a complete tool has been developed for automatic translation of POOSL specifications into C++.

Table of Contents

Table of Figures

1 Introduction

1.1 Project Context

The design of information-technology products is becoming more difficult as the complexity of those products grow. Products often contain a mixture of hardware and software components. The markets change quickly and lead to stringent time-to-market requirements. Customers request high reliability, high performance and low costs. This demands for new methods and tools that support adequate hardware/software analysis, specification and design so that the development effort can be minimised. At the section of Information and Communication Systems, Faculty of Electrical Engineering of the Eindhoven University of Technology, development of these methods and tools is subject of active research. This research has resulted in the Software/Hardware Engineering (SHE) method [PVS95][PV97].

The SHE method is an object-oriented method covering co-specification, analysis and design. It offers the formal specification language POOSL (Parallel Object-Oriented Specification Language) [Voe95a][Voe95b][PV97]. Because of the formal nature of this language, a POOSL system specification is unambiguous. Another advantage is that it is implementation independent. The SHE method also offers a set of behaviour-preserving transformations [PVS96][PV97]. Using these transformations, the architecture and topology of a system can be modified without changing the system's overall functionality. This way, architectural design decisions do not have to be made beforehand. The transformations allow these decisions to be made gradually during system analysis and design.

To offer system engineers a working environment for the SHE method, computer automated tools are essential. The tools should support the different development tasks such as transformation, verification, simulation and implementation.

1.2 Objective

The objective of this master's project is to develop a method for translating POOSL specifications into C++. The method has to be generic so that the translation process can be automated by a compiler. Initially, the goal was to develop a translation for the process part of POOSL and to focus on the inter-process communication in combination with the select, abort and interrupt statement and guarded statement. However, during the project the need arose to develop a translation for the data part also, since the way the data part is translated has a great impact on the translation of the process part.

The language C++ [Str92] has been chosen because it is one of the most widely used and accepted object-oriented programming languages in the industry. Furthermore, C++ compilers are available on a very large range of different computer platforms.

By working close together with the development of a POOSL compiler [Lei97], the joint effort results in an automated tool for implementing POOSL specifications into C++. This is also very useful for verifying the translation method, since the compiler takes over the time-consuming task to write test translations by hand.

This master's project comes down to developing a library (the C++ POOSL library) that contains enough functionality to implement POOSL specifications in C++. In combination with this library, translation-rules have to be developed that describe how the library must be used. Figure 1.1 shows the trajectory from POOSL to C++ and from C++ to executable program.

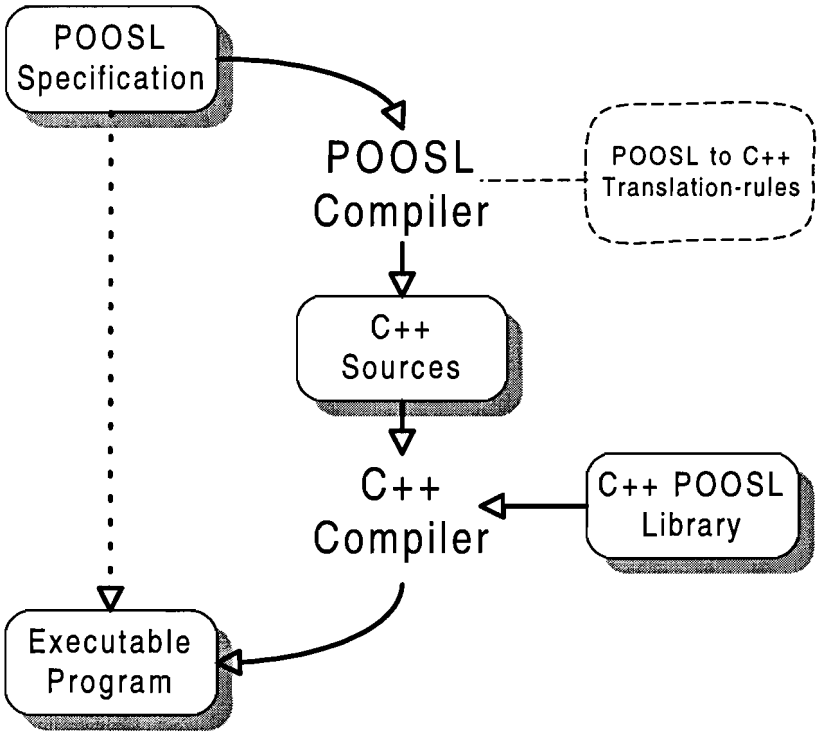


Figure 1.1: From POOSL to Executable Program

1.3 Thesis Organisation

This thesis is organised as follows:

- *POOSL Data Part to C++*. Chapter 2 describes the translation of the data part of POOSL. It explains the creation of data objects and reference counting technique for removing dispensable data objects. Further, the deepCopy procedure is explained which is used for making "deep" copies of data structures.

- *POOSL Process Part to C++*. Chapter 3 describes the translation of the process part of POOSL. It discusses the implementation of the select, abort and interrupt statement. Also discussed are the guarded statement and tail recursion.

- *The POOSL Library*. Chapter 4 deals with the POOSL library. This C++ library contains classes and functions that are required for the implementation of a POOSL specification in C++. All relevant classes are discussed. Comprehension of this chapter requires knowledge of C++.

- *Translation Example.* Chapter 5 uses some examples to describe how a POOSL data, process and cluster class must be translated into C++ classes. These examples are also used to explain implementation details. Like the chapter 4, comprehension of this chapter requires knowledge of C++.

- *From POOSL to Executable.* Chapter 6 is explains how to build an executable program from a POOSL specification in two steps. The first step is to compile the POOSL specification to C++ sources, and the second is to compile these sources and link them together with the POOSL library. The result is an executable program.

- *Conclusions, Results and Future Work.* Chapter 7 presents the conclusions, results and recommendations for future work.

2 POOSL Data Part to C++

2.1 Introduction

This chapter discusses the basics for implementing the POOSL data part in C++. A distinction is made between POOSL-primitive data classes, user data classes and user-primitive data classes. POOSL-primitive data classes are defined by the semantics of POOSL. User data classes are classes defined completely in POOSL and can therefore be generically translated into C++. User-primitive data classes differ from those in that they cannot be (completely) defined in POOSL and must therefore be written in C++ by hand.

2.2 Data Objects

In POOSL, data objects are instances of data classes. The behaviour of a data object is defined by its data class. Data objects (except for primitive data objects) are dynamically created at run-time by using the *new* statement. This means that the user is responsible for the instantiation of data objects. Unlike creation of data objects, POOSL has no way to let the user explicitly destroy data objects. Since the C++ implementation must be able to run on a machine with finite memory, dispensable data objects must be destroyed. Therefore, it is the C++ implementation's responsibility to trace and destroy unneeded data objects.

2.2.1 POOSL-primitive Data Classes

The smallest building blocks for forming complex data structures are the POOSL-primitive data classes: *Boolean*, *Integer*, *Real* and *Char*. These classes are defined by the semantics of POOSL, so the POOSL user can use them but can not modify them. For this reason, these classes are implemented in a C++ library that contains all the required functionality for translating POOSL specifications into C++ (the POOSL library).

The *new* statement does not exist for POOSL-primitive data classes. All possible instances of these classes already exist. When an instance is required, only the reference to the object is used. For example, consider the following statements:

```
a := 3
a := a + 3
```

In both statements, the literal integer is a reference to the same object with integer value 3.

Every instance of a POOSL-primitive data class is unique. In other words: two instances of a POOSL-primitive data class with equal properties refer to the same object. A possible implementation in C++ is to place a reference to every created POOSL-primitive data object in a list. When instantiating a new object, the list is searched for already containing the required object. If it does, a reference to the object is returned, otherwise, a new object will have to be created. A major disadvantage of this implementation is that for every object instantiation the list must be searched. Therefore, the solution presented in the next paragraph has been chosen.

In C++, a new object is created for every instance of a POOSL-primitive data object. No difference is made whether or not the object already exists. Consequently, the equality operation, denoted as ==, must be implemented according to the flowchart of Figure 2.1. Although this solution differs from the one in the previous paragraph, the behaviour observable by the user is identical.

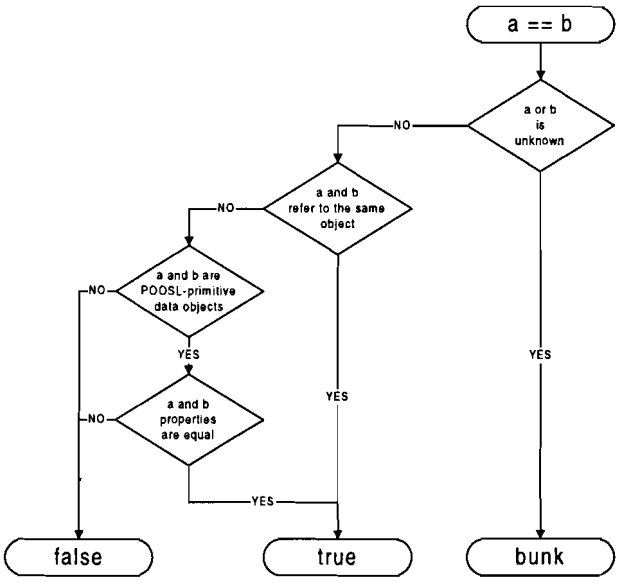


Figure 2.1: Equality (==) Operator Flowchart

The first test in this flowchart ("a or b is unknown") tests if at least one of the expressions evaluates to an unknown value (*bunk*, *iunk*, *runk* or *cunk*). The types of the expressions are not compared because it is a context condition that both expressions of the equality operator have the same type.

2.2.2 User Data Classes

The POOSL data part allows the user to define new data classes. These classes are called user data classes and can completely be defined using POOSL syntax. The translation of a user data class into C++ code is generic, which means that it can be compiled automatically without user interaction. The details of translating user data classes into C++ can be found in paragraphs 4.3.2 and 5.1.

2.2.3 User-primitive Data Classes

User-primitive data classes are data classes that contain one or more methods that cannot be defined using POOSL syntax. In POOSL, these user-primitive methods are only defined by their parameters and by a keyword indicating that the method is primitive. Obviously, the translation of a user-primitive data class into C++ is not generic and must therefore be done by hand. User-primitive data classes allow the use of C++ data types and operations like handling files, interfacing with external hardware, etcetera.

2.3 Garbage Collection

Garbage collection is the automatic reclamation of computer storage. In C++, the user must explicitly reclaim heap* memory by using a "free" or "delete" statement. POOSL however, does not have such a statement. The C++ implementation of a POOSL specification must therefore be equipped with a garbage collector that tracks down and deletes dispensable data objects. From several available garbage collection techniques, the reference counting method has been chosen. The reason for this is that reference counting is suitable for real-time systems and that it is relatively easy to implement. Its drawback is that it is not able to reclaim cyclic garbage, as will be explained in the next paragraph. However, implementing a real-time garbage collector without this deficiency would reach beyond the scope of this project.

2.3.1 Reference Counting

In a reference counting system, each data object is associated with a reference counter. The reference counter reflects the number of references (pointers) to the object. Each time a reference to an object is created, its reference count is incremented. When a reference to an object is destroyed, its reference count is decremented. This behaviour is integrated in the implementation of POOSL's assignment statement; a reference to the newly assigned object is created and the reference to the previously assigned object (a variable or parameter refers to *nil* object by default) is destroyed. When the reference count reaches zero, the object's memory will be reclaimed. Since the object itself may contain references, this may lead to the transitive decrementing of reference counts and reclamation of other objects. See Figure 2.2 for an example.

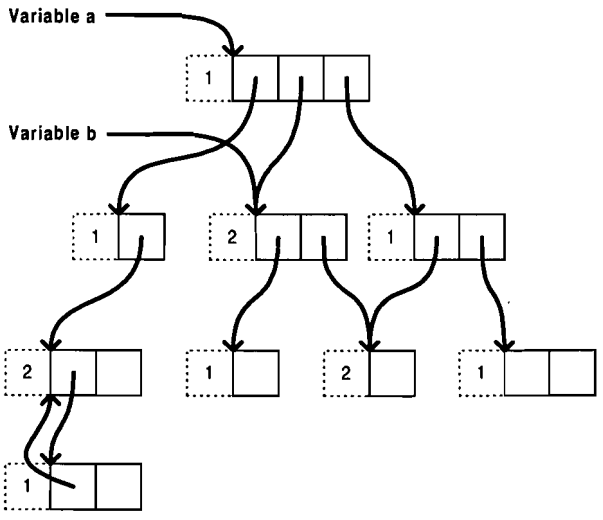


Figure 2.2: Reference Counting

An advantage of reference counting is its incremental nature, which means that the CPU-time spent on garbage collection is interleaved with the execution of the program. Because of the small interleave steps, the garbage collector will never interrupt the program execution for a long time. This is why reference counting is suitable for real-time systems. The term "garbage collector" further used in this thesis must be understood as an imaginary entity, because its implementation is interleaved with the translation of the POOSL specification.

* The term "heap" is used for the memory pool managed by a memory manager which allows dynamical allocation and deallocation of data objects.

A major disadvantage of a reference counting garbage collector is that it is not able to reclaim cyclic garbage. If the references in a group of objects create a cycle, the objects' reference counts will never reach zero. Consequently, the objects will not be reclaimed and remain as cyclic garbage. Figure 2.3 shows that after the deletion of variable a (see also Figure 2.2), the objects in the cycle refer to each other and therefore each has a reference count of one. Since there is no reference path from a variable to the cycle, the objects in the cycle can not be reached by the program. This means that the objects are garbage and must be reclaimed. However, the reference counting garbage collector fails to do so.

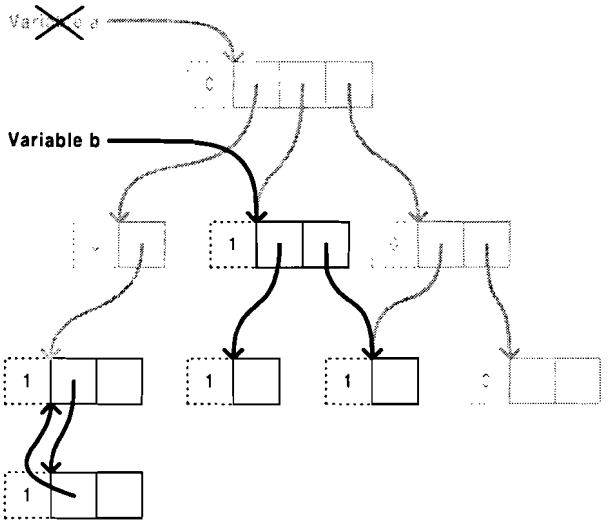


Figure 2.3: Cyclic Garbage

2.3.2 Other Garbage Collection Techniques

To solve the problem of cyclic garbage, other garbage collection techniques use a root set of the currently existing variables within the program. Instead of a reference count field, the objects have a tag field. At a certain event (a timer timeout, the amount of free memory passing a treshold, etcetera), the garbage collection algorithm is activated. The algorithm works like this: First, it untags all objects that are located on the heap. Then it tags all objects that can be traversed (directly or indirectly) from the root set. Finally, it reclaims all objects on the heap that are not tagged. The result is that the storage space of all the objects that cannot be reached from the root (which is garbage) is reclaimed. The problem with the algorithm in this form is that it is not suitable for real-time systems, because it interrupts the normal execution of the program for an undetermined time (depending on the amount of objects). More information on garbage collection techniques can be found in [Wil92] and [PS95].

2.3.3 Notifying the Garbage Collector

Whenever a reference to an object is destroyed, the possibility exists that the object has become garbage. In order to clean up the garbage, the garbage collector should be notified of such actions. It is therefore important to know where reference deletions take place. The assignment statement is the most obvious situation; the previously assigned object will no longer be referred by the variable or parameter in question. The other situation is the termination of a data or process method, because then the method's local variables and parameters are destroyed.

Care has to be taken with the data expression used as a statement. The user may use such a statement for its side effect. Evaluation of the expression can result in a new object with a zero reference count (take for example the statement *new(Complex)* instead of *a := new(Complex)* where *a* refers to the new *Complex* instance). The garbage collector needs to be notified of this. For this purpose, the POOSL library has a *Cleanup* function which destroys an object when its reference count is zero.

The data expression statement exp_a *method*(exp_b) is also a special case. Both expressions exp_a and exp_b can evaluate to a new object with a zero reference count (*(new(Complex) init(0,0)) add(new(Complex) init(1,1))* for example). If so, the objects may have become garbage items when the method terminates. The garbage collector needs to be notified of this also. More details will follow in paragraph 5.1.

2.4 DeepCopying

POOSL process instances communicate solely via a set of channels. This requires copying of all message parameters because shared objects introduce communication lines outside the set of channels. Since a superficial (shallow) copy of an object possibly shares sub-objects* with its original, the sub-objects must be copied also. The same holds for the copies of the sub-objects. The result of this recursive copy procedure is called a deepCopy. A deepCopy of a POOSL-primitive data object results in a reference to the object itself and not in a copy of the object. This does not give a sharing problem, since POOSL-primitive data objects do not contain instance variables and have no method that modifies the object itself. Hence, a POOSL-primitive data object cannot be changed, only the variable referring to it can be reassigned to another object. After reassigning the variable, the previously referred object is no longer shared so the problem does not exist.

Cyclic references in a group of objects introduce a difficulty when making a deepCopy. Every object must be copied only once and the copy of the group of objects must have the same cyclic references as the original. As a solution to this problem, each data object holds an internal reference field that is accessible to the deepCopy procedure. During the deepCopy procedure, the original object's reference field refers to the object's copy. When the object has not yet been copied, its reference field refers to nothing. Before copying an object, its reference field is checked to see if it has already been copied. If this is not the case, the object is copied and its reference field is updated. Otherwise, the reference is used to locate the copy. Figure 2.4 shows an example of how a deepCopy is made. Frame (A) shows the start situation where variable "a" refers to the data structure that has to be copied. In the following frames, the bold arrows indicate the reference path that has been traversed by the deepCopy procedure. A dashed arrow indicates that the original object's reference field refers to its copy. The meaning of colour of the objects is as follows:

* When an object has a reference to another object, the other object is called a sub-object.

- White indicates that the object has not been traversed yet.
- Grey indicates that the object has been traversed.
- Black indicates that the object and all its neighbours have been traversed.

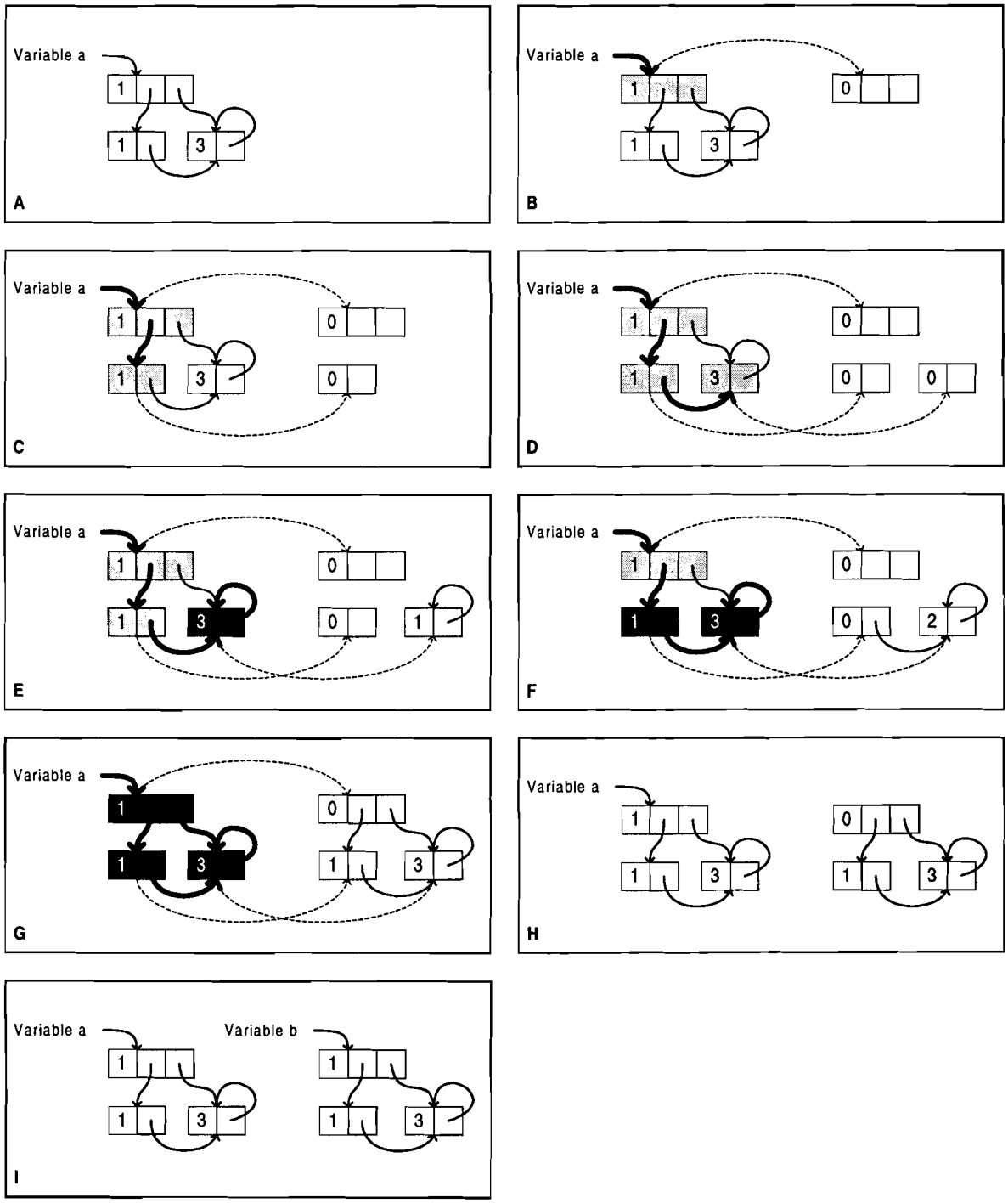


Figure 2.4: A deepCopy Example: b : = a deepCopy

At (A) the deepCopy procedure starts recursive copy procedure. At (B), this recursive copy procedure reaches the first data object and makes a copy of it. Since the copy is not yet referred by any variable, its reference count is set to zero. Next, it proceeds with all the

object's instance variables in a depth first manner. When an instance variable refers to a white data object, that object will also be copied by the recursive procedure. After that, the copied object will be assigned to the copy of the instance variable. At (E), the cyclic reference refers to an object that has already been traversed (and copied). Instead of copying it again, the recursive copy procedure locates the copy via its reference. When the complete data structure has been copied by the recursive copy procedure, the deepCopy procedure clears all reference fields (H). At the last step (I), the deepCopy of "a" is assigned to variable "b".

3 POOSL Process Part to C++

3.1 Introduction

Execution of process statements is not straightforward. The reasons for this are the select, interrupt and abort statements. An alternative of a select statement may only be chosen when the first statement of that alternative can be executed. If this first statement is a communication statement, the executability depends on whether a communication partner can be found or not. To make things even more complicated we can also add a receive condition to it or guard the communication statement with a guard condition.

The select statement introduces the possibility that a process must choose between multiple communication alternatives. When one or more of the alternatives can be executed, the process must make a non-deterministic choice and execute it. The process cannot possibly know which of the alternatives are executable because this depends on its communication partners. Even when there is a partner, this partner could run of with a third process. It is not [illegible] processes practice a trial and error strategy, so we must have some kind of

[illegible]n) statement can also be nested in an interrupt or abort statement. In the [illegible]*errupt* S_2), the interrupt status adds an extra condition to the executability of [illegible] and abort allow several statements to be active at the same time, each within [illegible]ironment. However, the statements do not execute in parallel because the [illegible]le each other. Nevertheless, we need some kind of context switching [illegible]tch statement execution between the local environments.

[illegible]cutes statements within its own environment and in parallel with the other processes. This also requires some kind of context switching mechanism. It does not require parallel statement execution because we can simulate parallelism by making a context switch between the environments of the processes after every executed statement.

Context switching should be done in combination with arbitration. This way, the arbiter can choose an environment which active statement is executable and switch to it. Prior to making such a choice, the arbiter must know the active statement of each environment. Therefore, execution of every active statement must be requested. Because the arbiter's task is to schedule and grant requests, it will be called the scheduler.

The alternatives of a select statement all share the same local environment. Execution of a select statement does not require parallelism because the alternatives exclude each other. Hence, we do not require context switching for implementing the select statement. However, by placing each alternative in a local environment, we can easily implement nesting. This is because an environment can take control over nested statements, which allows a divide and conquer strategy. For the same reason, a guarded statement is also placed in a local environment.

The context switching mechanism can be implemented in C++ by threads. For now, it is sufficient to know that a thread is a local environment that executes a sequence of statements. More information on threads will follow in paragraph 4.2.

3.2 Requests

A process that wants to execute a statement must first get permission from the scheduler. Permission can be requested by submitting a statement specific request. After requesting, the process must wait until the scheduler grants the request. There are several types of requests and each has a special purpose. All types of requests and request flows between processes, channels and the scheduler are illustrated in Figure 3.1.

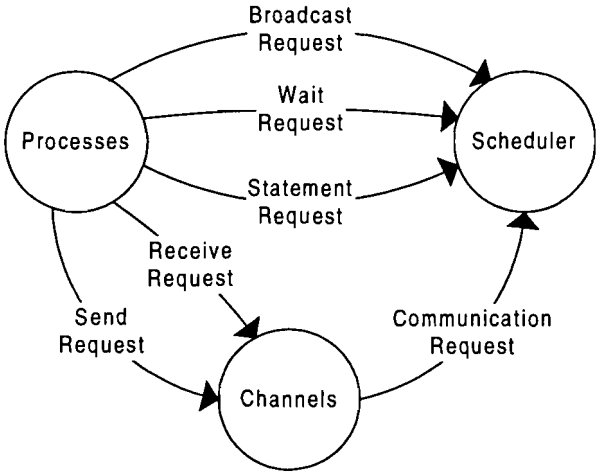


Figure 3.1: Requests and Request Flows

A statement request is used for requesting a process execution step such as a method call, data statement, and evaluating an if or while condition. Execution steps within data class methods (data execution steps) are always directly executed without requesting of any kind. This is possible since data execution steps can not be interrupted or aborted.

A send or receive request is used for requesting a rendezvous with another process connected to a specific channel. Each time a channel receives a request, it tries to find communication partners for that request. Every matching send/receive pair is formed into a communication request and submitted to the scheduler. In case of a conditional receive statement, the receive request contains the condition. This way, the scheduler can evaluate it to test whether it is executable* or not (see paragraph 3.6.1).

A broadcast request is used for requesting to send a message to all other processes that are waiting to receive that that message on a specific channel. Unlike a send request, a broadcast request is directly sent to the scheduler. This is because a broadcast statement will not wait for receivers. When there are no receivers, the broadcast statement just continues without sending a message. Communication partners are matched by the designated channel. Matching is initiated by the scheduler when granting the broadcast request.

* With "executable request" we mean that the statement requested by that request is executable.

A communication request is used for requesting a send or receive statement. Such a request is generated by a channel that detected a possible rendezvous between two processes. It contains a reference to the send and receive request that form the rendezvous.

A wait request is used to request an execution delay. A process that wants to suspend execution for some amount of time must send a wait request to the scheduler.

3.3 Process Tree

The execution of a statement in one environment (a thread) may interrupt, abort or (in case of a select) exclude execution of statements in other environments. To describe these kinds of environment relationships we use a tree structure called a process tree. Take for example the select statement. Each alternative will be executed by a separate thread. Once a choice is made, the chosen thread will continue and the other threads will be cancelled. This illustrates that the threads of a select statement are not independent. The threads of an abort or interrupt statement also have a special relation. For example: when executing an interrupting statement, execution of the interrupted statement is suspended until the interrupting statement ends. The relation of the threads in use by a process can be described in a tree structure. This tree structure is called the process tree. The real execution of statements is done by the leaf nodes. The select, abort, interrupt and guard nodes only control their sub-nodes. All sub-nodes notify their parent about their actions. This way the parent can take the appropriate control action. See Figure 3.2 for a process tree example.

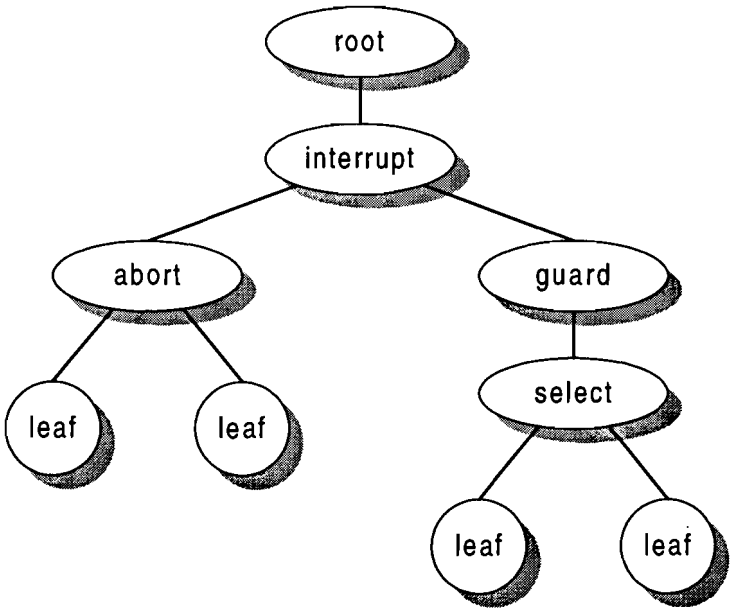


Figure 3.2: Process Tree Example

In the C++ implementation, every POOSL process has its own process tree. The process tree changes dynamically as the process executes its statements. The root node has no other function than to indicate the root (or top) of the process tree. Every other node reflects a POOSL statement of the process. Each node of the process tree (except for the root) is associated with a thread that does the actual statement execution.

3.3.1 Leaf Node

A leaf node corresponds to a composition of process statements in the POOSL specification. Execution of those statements is done by the thread associated with the leaf node. When a statement of a composition is a control* statement (a select, abort, interrupt or guard), the leaf node is transformed into a control node. The control node's task is to create and control its sub-nodes. After termination of the control statement, the control node is transformed back into a leaf node and resumes execution of the composition. A leaf node terminates when the last statement of the composition has been executed.

An example:
Let a composition of statements S be S_1; (S_2 *abort* S_3); S_4. The leaf node corresponding to S executes† S by subsequently executing S_1, (S_2 *abort* S_3) and S_4. When executing (S_2 *abort* S_3), the leaf node is transformed into an abort node. After that, the abort node creates two sub-nodes (one leaf node for S_2 and one for S_3) and takes control over them. The execution of statements S_2 and S_3 is performed by the threads of the sub-nodes. When S_2 or S_3 terminates, the sub-nodes are killed and the abort node itself is transformed back into a leaf node that continues S by executing S_4.

Except for control statements, all process statements are execution steps and must therefore be granted by the scheduler prior to execution. Therefore, the leaf node has to request the action it wants to perform.

3.3.2 Select Node

A select node is used to implement the select statement (also called choice statement). It does so by creating a sub-node for every choice. When a sub-node of a select node is selected to perform a first process execution step, all the other sub-nodes are killed and the selected sub-node continues executing its statements. The select statement terminates when the selected sub-node has executed its last statement.

3.3.3 Abort Node

An abort node is used to implement the abort statement. It does so by creating one abortable sub-node and one aborting sub-node. When the abortable sub-node performs an execution step, nothing special happens. However, when the aborting sub-node performs an execution step, the abortable sub-node is killed and the aborting sub-node may continue its execution. The abort statement terminates when one of its sub-nodes has executed its last statement.

3.3.4 Interrupt Node

An interrupt node is used to implement the interrupt statement. It does so by creating one interruptable sub-node and one interrupting sub-node. When the interruptable sub-node performs an execution step, nothing special happens. However, when the interrupting sub-node performs an execution step, execution of the interruptable sub-node is suspended until the interrupting sub-node terminates. On termination of the interrupting sub-node, the interrupting sub-node is recreated. This way the interruptable sub-node can be interrupted multiple times. The interrupt statement only terminates when the interruptable sub-node has executed its last statement. An interrupt node can suspend the execution of its sub-node by

* Because the select, abort, interrupt and guard take control over statements, they are called control statements
† With execution of statements by a node is meant the execution of statements by the node's thread.

setting the sub-node's interrupt flag. For this reason, all nodes have such a flag. The interrupt flag is used by the scheduler to see if a request is executable. More on this will follow in paragraph 3.6.1.

3.3.5 Guard Node

A guard node is used to implement the guard statement. It does so by creating a sub-node for the guarded statements. The guard node has a reference to the guard condition. Before the guarded statement may perform a first execution step, the guard condition is evaluated. When the evaluation results in true or bunk, the guard condition is discarded and the sub-node starts execution. Otherwise, nothing happens. The guard statement terminates when the sub-node has executed its last statement.

3.4 Tail Recursion

A process method can be called recursively or tail-recursively. Tail recursion differs from normal recursion in that it does not build up the call stack. Another special property is that it prevents building up the process tree which means that the context of the call will be lost (see next example). To avoid a call stack overflow, a process method call should be translated to a tail-recursive call whenever possible.

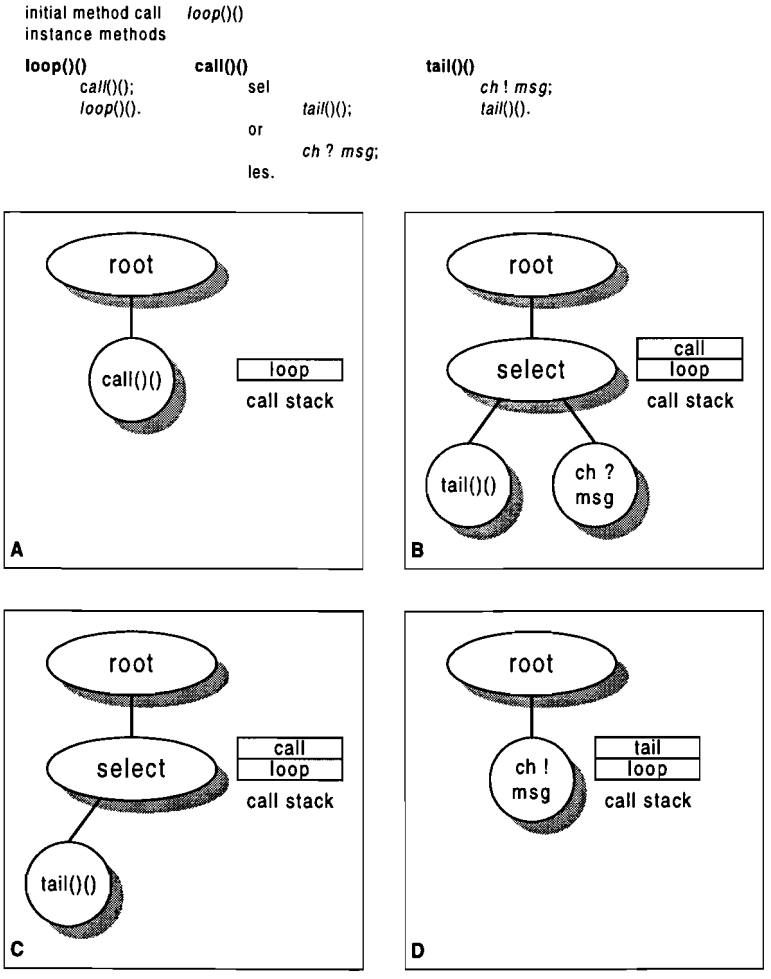


Figure 3.3: Tail Recursion, Call Stack and Process Tree

When a method is called upon tail-recursively, the top of the call stack is removed and the process tree is trimmed to the node from where the latest recursive call has been made. This illustrated in Figure 3.3. In situation A, initial method 'loop' has been called recursively. From there, method 'call' is also called recursively, which leads to situation B. In situation B, the select statement has transformed the leaf node into a select node with two alternatives. In situation C, the first alternative of the select statement has been chosen. The following step is a tail-recursive call to method 'tail'. In situation D, the process tree has been trimmed to the node from where the latest recursive call was made. In addition, the top of the call stack has been removed prior to calling method 'tail'.

A method may be called tail-recursively except for the following situations:
- when the method call is followed by a statement
- when the method has return parameters
- when the method is called from a method that has return parameters
- in the statements of an interrupt
- in the abortable statements of an abort
- in a while loop

When a method call is followed by one or more statements, the method can not be called tail-recursively. This is obvious since otherwise the statements will never be executed. When a method or a calling method has return parameters, tail recursion is again not possible since the return parameters need to be returned.

Unlike the select statement, the context of an interrupt statement must be preserved until it is terminated. Because tail recursion destroys the context, it can not take place in the statements of an interrupt.

When an aborting statement is executed, the abortable statements are discarded and the aborting statements are executed as if there was no abort statement. The context of the abort node may therefore be lost. This is not true when an abortable statement is executed. Hence, tail recursion in an abort is only possible in the aborting statements.

Tail recursion can not take place in a while loop because tail recursion destroys the context which would result in termination of the loop, disregarding the while condition.

3.5 Channels

In POOSL, processes communicate by sending messages over communication channels. The interconnection structure of these processes and channels is described by a behaviour specification. In C++, a POOSL channel is represented by one or more channel objects (see Figure 3.4). Such a channel object is owned* by a cluster and can be connected to several processes and/or sub-clusters of that cluster. A channel object can only exist within its cluster. When it wants to communicate with the outside of its cluster, it must pass its message request to another channel object via a cluster connector. The channel structure in POOSL is hierarchical. In C++, this means that a channel is restricted to have at most one connection to

* A channel object is owned by the lowest-level cluster containing that channel object. For uniform translation into C++, the top-level objects (channels, processes and/or clusters) are also placed in a cluster (called "System").

its own cluster. We will take advantage of this restriction when passing message requests over channels. Channels are connected to processes via process connectors the same way as they are connected to cluster connectors.

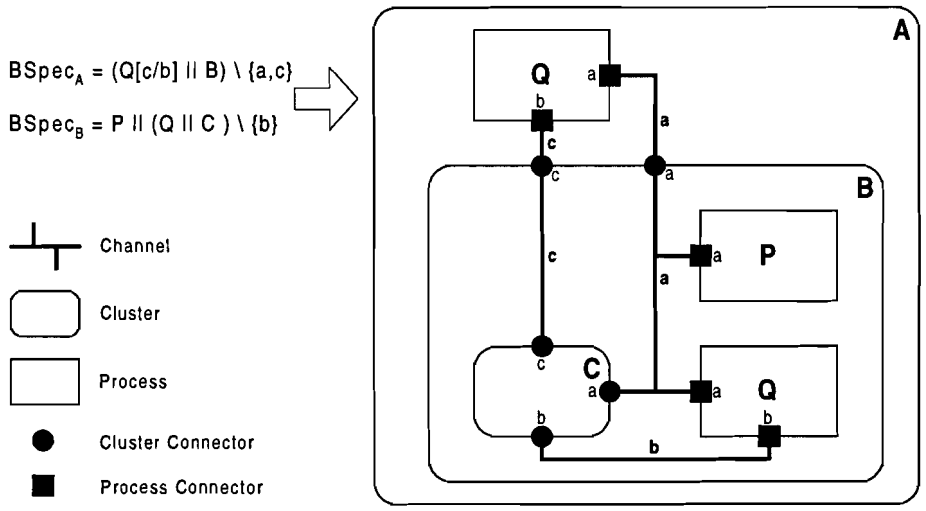


Figure 3.4: Translating a Behaviour Specification to C++

3.5.1 Message Request Passing/Storing

A POOSL process that wants to communicate with another process over a specific channel sends a message request to that channel. At that time, there may not be a suitable communication partner present. However, it is possible that one or more partners will present themselves later. For this reason, a message request is stored by the channel to which it is sent. As described in paragraph 3.5, a POOSL channel is translated into one or more C++ channel objects connected by cluster connectors. The channel structure is hierarchical. All channel objects (except for the top-level object) pass their incoming message requests up into the channel hierarchy. The top-level channel object stores each incoming request into a buffer and tries to find communication partners for it.

A process connector is a channel access point for a process. To send a message request to a channel, the process must send the request to the appropriate process connector. For each channel in a communication channels section of a POOSL process, there exists a process connector object in the C++ implementation.

A cluster connector is used for passing up message requests in the channel hierarchy. A channel object may be connected to several cluster connectors of lower-level clusters. Because of the hierarchical channel structure, a channel object may have only one connection to a cluster connector on the edge of its cluster. For each channel in a communication channels section of a POOSL cluster, there exists a cluster connector object in the C++ implementation.

3.5.2 Matching Communication Partners

Each time a top-level channel object receives a message request, it stores the request in a request buffer. After that, the channel object searches its buffer for possible communication partners. A partner is found when the following conditions are met:

- One request must be a send request and the other a receive request.
- Both requests must have the same message identifier.
- Both requests must have the same number of parameters.
- Both requests must originate from different processes.
- Both requests may not have passed different cluster connectors of the same cluster.

The first three conditions are obvious. The second last condition is according to the semantics of POOSL that a process can not communicate with itself.

The last condition prevents additional cluster behaviour introduced by channel connections outside a cluster. See Figure 3.5 for an example. In situation A process P and Q can not communicate with each other since there exists no channel between them. In situation B, process P and Q are allowed to communicate with R. Channel object d also connects process P and Q together but they are not allowed to communicate with each other. The reason for this is that otherwise the internal cluster behaviour can be affected by external channel connections, which is in conflict with the compositional nature of POOSL. In general, when two message requests have passed different cluster connectors of the same cluster, they are not allowed to form a communication pair.

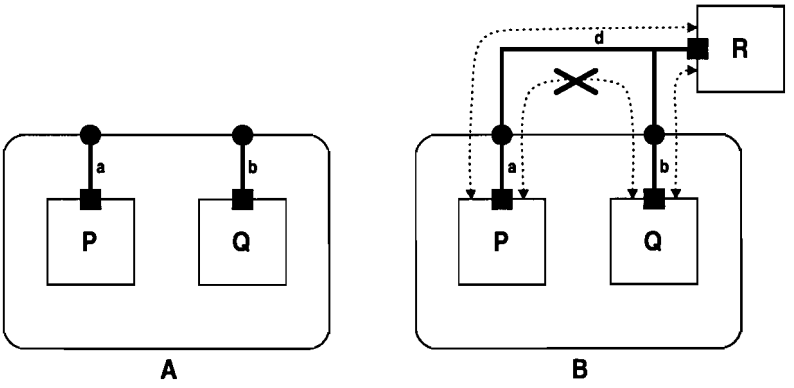


Figure 3.5: Prohibited Additional Cluster Behaviour

3.6 The Scheduler

The scheduler controls the execution of POOSL statements by operating in two phases. First, it collects all requests that are submitted by the threads of the process tree nodes and by the channels. After that, the scheduler non-deterministically selects a request and grants it.

In POOSL, the delay statement is used to indicate an amount of time passing by. Other statements do not consume time. A POOSL program must make maximal progress, which means that statements that do not cost time to execute must be executed first (which complies with the semantics of the delay primitive [Gei96]). Consequently, non-delay statements have precedence over delay statements. A delay request may only be granted when there are no executable non-delay requests remaining. When there are no executable requests left (delay or non-delay), the scheduler run ends which indicates that the POOSL program has terminated or is in deadlock. For collecting requests, the scheduler maintains two lists; one for the delay requests and one for the non-delay requests. The reason for this will be apparent after reading the next paragraph.

3.6.1 Granting non-delay requests

After selecting and before granting, a non-delay request is tested to determine whether it is executable or not. A request is executable when it is neither blocked by a guard nor suspended by an interrupt. This is tested by traversing the path from the requesting node to the root. When one of these nodes has a set interrupt flag or is a guard node with a condition that does not hold, the request is not executable. In addition, when it is a receive request with condition, its condition must hold also. Further, a communication request is executable when its send and receive request are executable. When a selected request is not executable, it is temporarily removed from the list of scheduled requests and an another attempt is made to select an executable request. This procedure repeats until an executable request has been selected or until there are no requests left. In the first case, the request will be granted. In the second case, an attempt is made to grant a delay request.

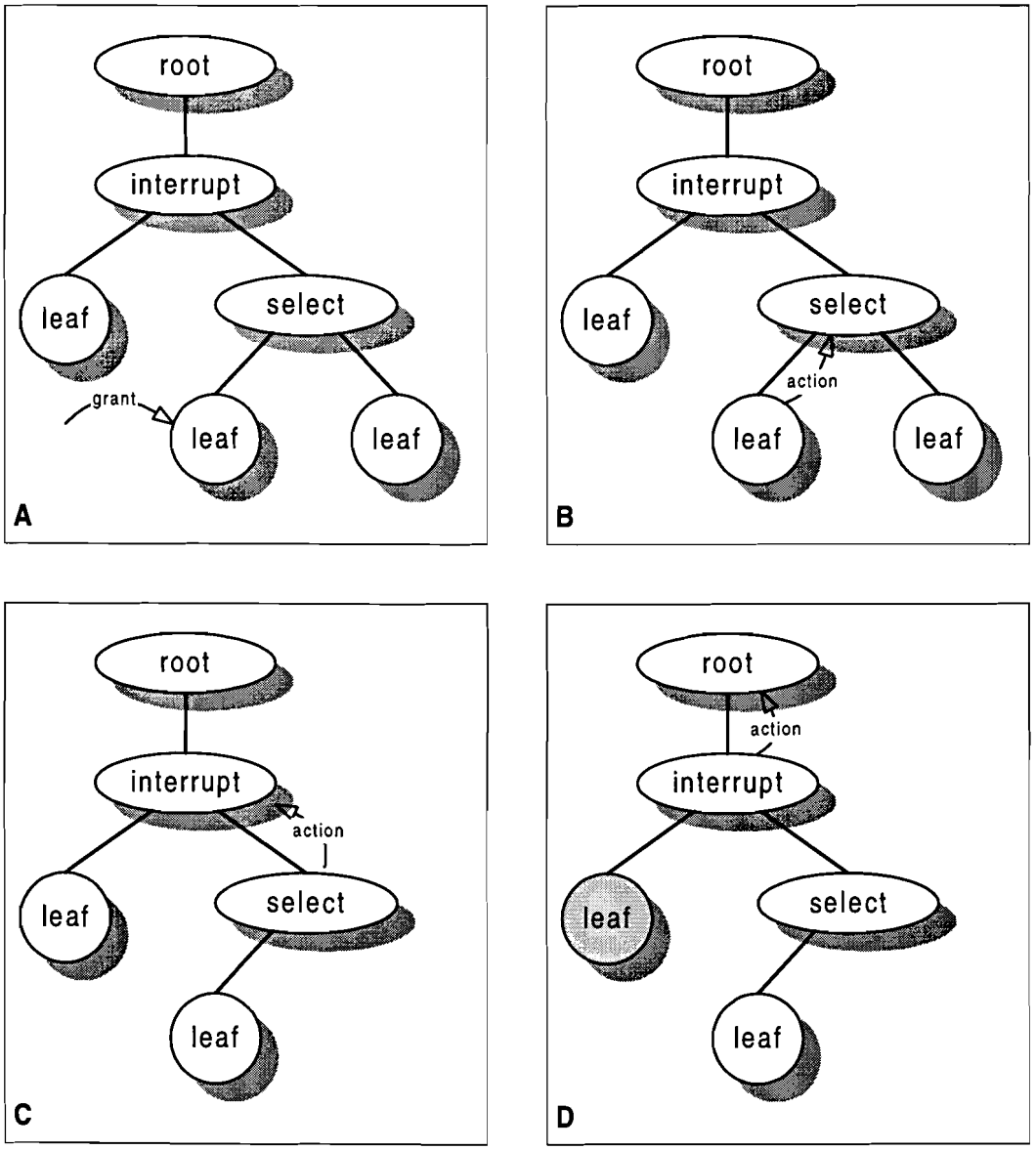


Figure 3.6: Granting

Granting a non-delay request starts a procedure that traverses a whole path in the process tree from the requestor (a leaf node) to the root. Each traversed control node in that path is notified about which sub-node is going to perform a process execution step (an action). When notified, a control node must take the appropriate control action depending on its node type and state as previously described in paragraph 3.3. Figure 3.6 illustrates the granting procedure. Frame (A) shows granting of the leaf node's request. The leaf node has now permission to perform an execution step and notifies its parent about this (B). The parent (the select node) reacts by killing its other child. Then, the select node notifies its parent (the interrupt node) about the action (C). The interrupt node's action is setting the other child's interrupt flag (indicated by the gray colour) and notifying the root node (D). This means that the request submitted by that child is now not executable. The child's interrupt flag will be cleared after the other child (the select node) has terminated. The root node will take no further action and the scheduler will make a context switch to the local environment of the granted node. The node will then execute its granted statement and request its next statement. If there is no next statement, the node will terminate.

All nodes that submit a request keep a reference to that request. This way, the node can revoke its request when the node is killed by its parent. Except for send and receive requests the requests are directly revoked at the scheduler. The send and receive requests are revoked at the channel to which they were sent. The channel must then revoke all communication requests that have a reference to the request in question. This is done by searching all the scheduled communication requests.

3.6.2 Granting delay requests

The delay statement *delay(n)* takes n units of POOSL time to execute. The POOSL time remaining to execute the delay statement decreases during its execution. When a delay statement is not executable, (when it is interrupted or blocked by a guard) no POOSL time is spent on it. In a simulator environment, the POOSL time does not have to be synchronised with the wall clock time. Therefore, instead of waiting for the time to pass by, we can make a leap in time and after that, grant the delay requests that have no time remaining. When making such a time leap, the remaining time for every executable delay request must be decreased by the leap size. In a real-time environment, the same strategy can be taken. However, just after making a time leap the scheduler must wait to synchronise the POOSL time with the wall clock time. The current version of the scheduler (implemented in the POOSL library) supports only simulated time.

According to the semantics of the delay statement, execution of a delay is not a process execution step. Therefore, the process tree is not affected by granting a delay request unless it is the last statement of a node, which causes the node to terminate. The only result granting a delay request has, is that its requestor (a leaf node) may continue by requesting the next statement (or terminate when there is no next statement).

4 The POOSL Library

4.1 Introduction

In this chapter, the POOSL library will be explained. This C++ library contains classes and functions that are required for translation of POOSL specifications into C++ programs. All relevant classes are listed in appendix A. The C++ translation of a POOSL specification consists of the POOSL library and a set of C++ source files. Each source file represents a POOSL class (data, process or cluster). All source files compiled and linked together with the POOSL library result in an executable program that represents the POOSL specification. How the set of source files must be generated out of a POOSL specification is defined in appendix B. To automate the translation process, a POOSL compiler has been developed [Lei97].

4.2 Threads

4.2.1 Why Threads?

Within the context of a C/C++ program, a thread is a lightweight process that executes a function in (pseudo) parallel with other threads. The threads can communicate with each other via shared memory. Threads can be easily created and terminated, and with less operating system effort than normal processes.

The parallelism in POOSL can be implemented in C++ by using threads. As described in [Fel96], threads are more suitable for this task than processes are, because threads use less memory and other system resources.

4.2.2 DCE Threads

The DCE (Distributed Computing Environment) thread library, available for the HP UX 9.05 system enables the C/C++ programmer to use threads. The library interface functions are derived from the IEEE POSIX (Portable Operating System Interface) standard 1003.1c. For more ease of use, the POOSL library has been equipped with a set of C++ classes that act as a C++ layer around the POSIX interface routines. The most important features of the thread library will be explained in the next paragraphs. The POOSL library can be easily ported to any computer platform that supports these features. For more information on the DCE thread routines see [OSF95].

4.2.3 Semaphores

All threads share the same memory pool (except for the stack). A synchronisation mechanism is required to prevent conflicts when two or more threads simultaneously try to modify a shared variable. The POSIX interface routines provides mutexes and condition variables for this purpose. For more information on this see [OSF95]. Both facilities are used by the *Semaphore* class in the POOSL library, which provides a more friendly way to use synchronisation. The *Semaphore* class supports a *Wait* and a *Signal* function. In the POOSL library, the *Wait* function is used when a thread wants to wait for a signal. The waiting thread is blocked until another thread releases it by sending it a signal (with the *Signal* function). A thread will not be assigned to a processor as long as it remains blocked.

4.2.4 Scheduling Policy

The POSIX interface routines support several scheduling policies. The POOSL library only uses the FIFO (First In, First Out) policy. FIFO scheduling is non-preëmptive. This means that only one thread is running at a time and that it must explicitly release the processor to give the next queued thread a chance to run. When a thread blocks, the processor is automatically released. Another way to release the processor is to execute the *Yield* function. An advantage of the FIFO scheduling policy is that the programmer can define the context switching points and keep these out of critical sections. A disadvantage is that it can not take advantage of a multiprocessor system because that would require multiple threads to be running at the same time.

4.2.5 Thread Priority

FIFO scheduled threads can be given different priorities. The thread with the highest priority runs until it blocks. If there are more threads with the same priority, and that priority is the highest among other threads, the first thread to begin running continues until it blocks.

4.2.6 Creating, Cancelling and Detaching a Thread

At creation of a thread, a start-up function must be supplied. This function, including any nested function invocations will be executed when the thread is running. The thread terminates when the function returns or exits or when the thread is cancelled (*Cancel*). The POOSL library uses the synchronous cancellation scheme. This means that a cancelled thread defers termination until it reaches a specific cancellation point (a *Wait* function call, for example). A terminated thread still occupies system memory (for its stack and thread descriptor). This memory must be reclaimed by detaching (*Detach*) the thread. A thread can also be detached before it has terminated. If so, the memory will be reclaimed as soon as the thread terminates.

4.2.7 Joining a Thread

If a thread wants to wait for another thread to terminate, it can join that thread by calling the *Join* function. This has the result that the joining thread blocks until the joined thread is terminated. When the joined thread terminates by returning a pointer, that pointer will also be returned by the *Join* function. In other words, the joining thread receives the object that the joined thread returns (by a pointer). Joining a thread is possible as long as it has not been detached.

4.3 C++ Library Classes

4.3.1 POOSLObject Classes

The POOSLObject classes define the objects that are required for the C++ translation of a POOSL behaviour specification (see paragraph 3.5). Figure 4.1 gives an overview of these classes together with their hierarchical structure.

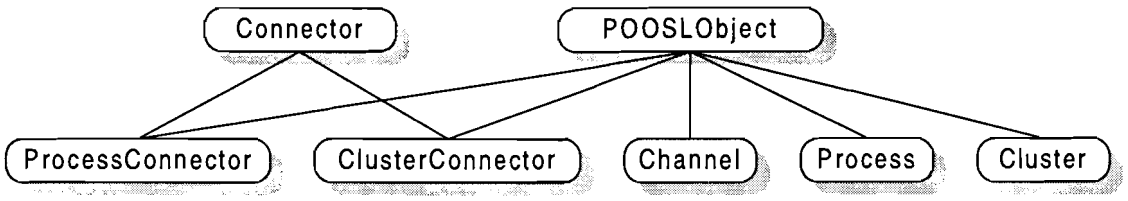


Figure 4.1: POOSLObject classes

The POOSL user may define process and cluster classes at will. These POOSL classes are translated to C++ classes that are derivatives of the *Process* or *Cluster* class. This way, the behaviour that is shared by all process and cluster classes is inherited by the user-defined classes. *ProcessConnector* and *ClusterConnector* objects are used to connect *Channel* objects to user defined process and cluster objects respectively.

All classes, which objects are data members* of a user defined process or cluster class, inherit the POOSLObject class. *ProcessConnector* objects are data members of user defined process objects. *Channel*, *Process*, *Cluster* and *ClusterConnector* objects are data members of user defined cluster objects. The POOSLObject class gives its inheritor's objects a way to know their owner†.

4.3.2 DataObject Classes

The *Boolean*, *Integer*, *Real* and *Char* classes are the C++ counterparts of the POOSL-primitive data classes. A defined POOSL data class must be translated into a C++ class that is a derivative of the *NonPrimitiveDataObject* class. This way, the behaviour shared by all user-defined classes (reference counting and deepCopying) is inherited. All data classes inherit the *DataObject* class that defines behaviour like reference counting (garbage collection), deepCopying, variable assignment and the equality operation (==). Figure 4.2 shows an overview of the DataObject classes.

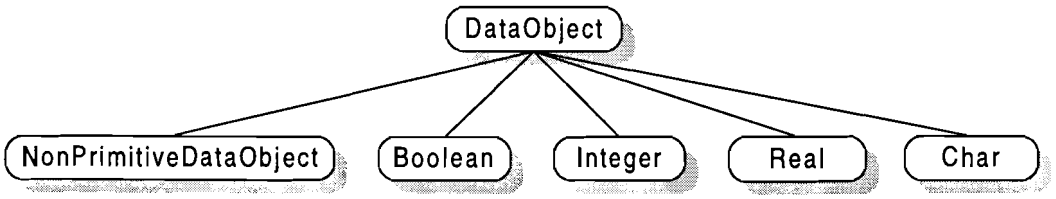


Figure 4.2: DataObject Classes

The POOSL library contains a static object for the unknown value (*bunk*, *iunk*, *runk* and *cunk*) of each POOSL-primitive data class. These objects are special instances of the *Boolean*, *Integer*, *Real* and *Char* class respectively. Since the *nil* object has no type, it is represented by a special instance of the *DataObject* class. Obviously, none of these special instances may be reclaimed by the garbage collector. Therefore, each of these instances has been given an initial reference count of one (instead of zero). This way the garbage collector will never treat them as garbage.

* An instance variable of a C++ class is called a data member.
† A POOSL specification has a static and hierarchical tree structure of processes, clusters and channels. The hierarchical parent of an object in that tree is called the owner.

4.3.3 StackElem Classes

In POOSL, a process method is able to return multiple parameters. However, in C++ a member function* can only return a single parameter. A solution to this problem is to make use of an array of *DataObject* objects. Multiple parameters can be put into a dynamically created array and then be returned as one parameter. Upon reception, the receiver must bind the array's elements to its receiving variables and then destroy the array.

A process method that does not have return parameters can make a tail-recursive call. The implementation is as follows: First, the calling method creates a *Tail* object containing the tail-recursive method and its input parameters. Then, the calling method terminates by returning the *Tail* object. Finally, the *Tail* object is used to execute the tail-recursive method.

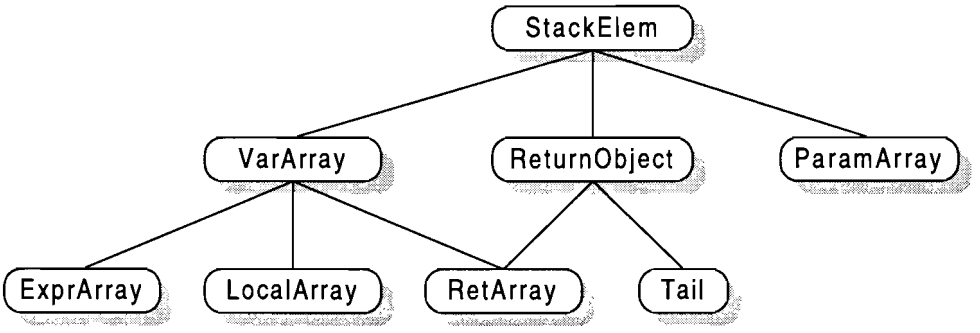


Figure 4.3: StackElem Classes

The C++ translation of a POOSL method is a member function with *ReturnObject* as return type. This enables the member function to return either a *RetArray* or a *Tail* object depending on whether it wants to return with parameters or make a tail-recursive call. *RetArray* and *Tail* objects are processed by a routine that initially executes a recursive method call. Figure 4.4 shows a flowchart of this method calling routine.

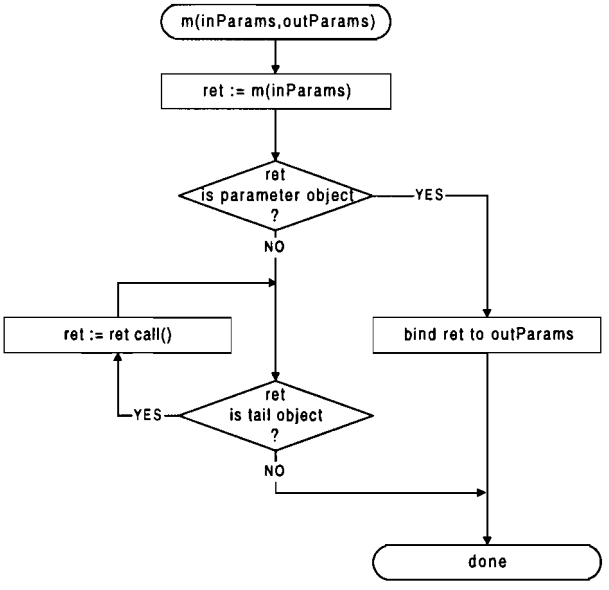


Figure 4.4: Method Calling Routine

* C++ class methods are called member functions.

The data object reclamation routine (garbage collection) and the deepCopy routine must have access to the instance variables of the objects they are processing. Both routines are defined by the *NonPrimitiveDataObject* class, so that they are inherited by all user defined data classes. To store the instance variables, the *NonPrimitiveDataObject* class has a *VarArray* data member. Variations on the *VarArray* are the *ExprArray* and *LocalArray*. Both have a special purpose. The *ExprArray* is used for passing data expressions when calling a method and when sending or broadcasting a message. The *LocalArray* is used for storing a method's local variables.

Received message or output parameters need to be bound to variables. Parameters are always sent using a *VarArray* object because is offers a uniform way for transferring one or more parameters at once. In order to bind the parameters to their variables, the locations (addresses) of the variables must be known. This is why we need the *ParamArray* class. An object of the *ParamArray* class can store multiple variables by their reference. All the parameters in a *VarArray* object can be bound to the variables of a *ParamArray* object. Each parameter in the *VarArray* will then be bound to the corresponding variable in the *ParamArray*.

All classes inherit the *StackElem* class. Objects of a *StackElem* class can be put onto a stack. This is necessary because all of the above objects are created dynamically (are placed on the heap) by threads. The thread that creates an object is also responsible for destroying it when its no longer needed. Unfortunately, it is possible that the thread is killed after creating an object but before destroying it. To keep track of such objects, they must be pushed onto the thread's cleanup stack. Objects on the cleanup stack that have become dispensable must then still be explicitly destroyed by the thread. However, when the thread is killed, the process tree node takes care of this by destroying all the objects located on the cleanup stack.

A more elegant solution would be to place the objects "by value" on the thread's data stack. Normally, when a C++ data stack is removed, the deconstructors of the objects on the stack are executed to let the objects deconstruct themselves. However, the third party thread routines do not support this since the data stack of a killed thread is simply removed without deconstructing its objects.

4.3.4 Request Classes

Figure 4.5 shows an overview of the Request classes. Most kinds of requests have already been explained in paragraph 3.2.

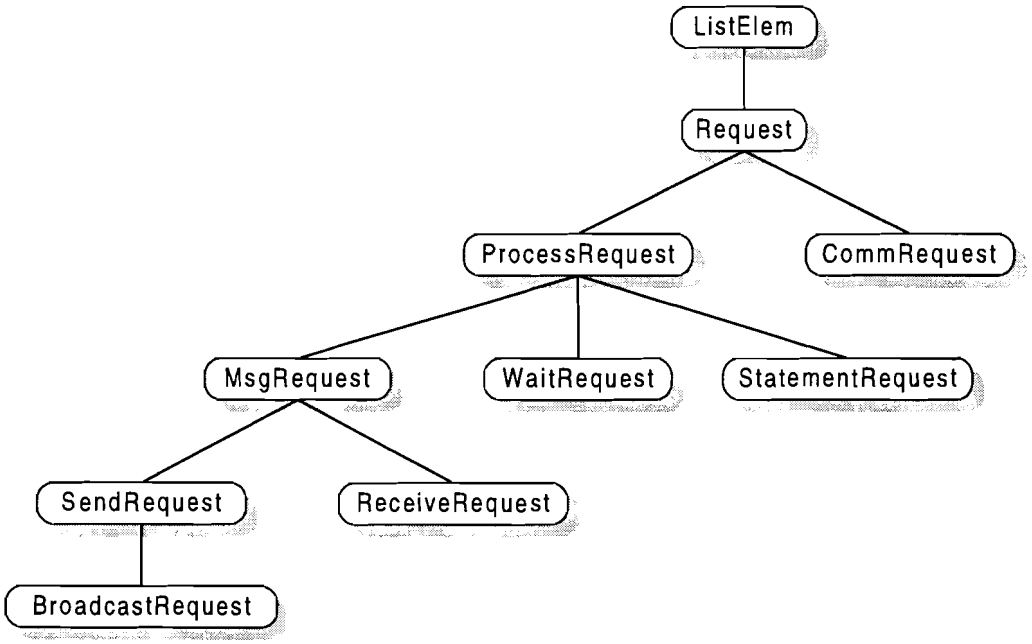


Figure 4.5: Request Classes

All classes inherit the *ListElem* class so that the objects of these classes can be put into a list. This is necessary to store and process the requests (by the channels and the scheduler). The *ProcessRequest*'s all have in common that they are generated by processes.

4.3.5 Scheduler and ProcessNode Class

The scheduler and the process tree nodes are the objects that perform the actual program execution. Every leave of the process tree requests its POOSL statement and waits for the scheduler. The scheduler selects a request and grants it. The scheduler and the process tree nodes execute in parallel. That is why the classes of these objects are inheritors of the *Thread* class (see Figure 4.6). The *ProcessNode* class is also a *ListElem* class. This allows that process nodes can be placed in a list, which is necessary for "hanging" multiple children nodes (sub-nodes) under a parent node.

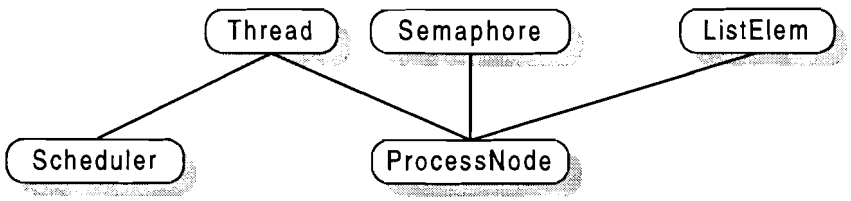


Figure 4.6: Thread and Semaphore Classes

The scheduler thread and the process node threads have to be executed in a specific sequence. First, the scheduler blocks, the process trees build up and the leave nodes submit their requests. After that, every node will block until its request (in case of a leaf node) is granted by the scheduler (or until the node is killed). Then, when all nodes are blocked, the scheduler continues by granting a request and signalling the node(s) that may continue executing the requested statement. Finally, the scheduler blocks again and the signalled nodes continue so that the sequence repeats itself.

After a control node has started up its children, it will block by waiting on its semaphore. When a child (except for the interrupting child of an interrupt node) successfully terminates, the semaphore is signalled so that the control node can continue. The control node then transforms back into a leaf node and starts with requesting the next statement.

The scheduler may not start selecting a request before all requests have been submitted. Therefore, two priority levels are used in combination with a FIFO scheduling policy. A low level for the scheduler thread and a high level for the process node threads. This way, the scheduler thread automatically continues when all process node threads are blocked.

4.3.6 Condition Class

The conditional receive statement and the guarded statement make use of a condition. To test whether such a statement is executable or not, the scheduler must evaluate the condition. A condition is a Boolean expression that can be made up of variables and parameters that are visible within its scope. Therefore, besides the member function that represents the condition, every instance of the *Condition* class is supplied with those variables and parameters. The condition itself is a member function of the user defined process class in question. Evaluation of the condition comes down to calling the member function with the variables and parameters as arguments.

4.4 Implementation Details

4.4.1 Thread Stacksize

The stacksize of the scheduler thread and the process node threads is internally fixed by the POOSL library. However, it is possible that these values need to be modified because they depend strongly on the POOSL specification. A value too large will result in excessive memory usage and a value too small will result in a system crash because the thread library does not support stack overflow checking. The appropriate value should be determined by experimentation.

4.4.2 Passing Arguments to Threads

The *MethodParam* and *BlockParam* classes (listed in appendix A) are used to pass the necessary arguments to thread functions. The thread functions only allow passing one pointer argument. By passing a pointer to a *MethodParam* or *BlockParam* object, we can achieve the desired result.

4.5 Performance Enhancements

4.5.1 Condition Evaluation Shortcuts

The possibility exists that the scheduler wants to evaluate a condition that has already been evaluated before. A re-evaluation would not be necessary if the condition's dependencies have not changed. It would then be sufficient to know the previous result so that a shortcut can be taken. However, keeping track of the dependencies probably costs more computing time than is gained by it.

The solution that has been implemented in the POOSL library is the following: Each process maintains a list of local *Condition* objects that have been evaluated by the scheduler so far. Each listed condition stores its last evaluation result so that re-evaluation is not necessary. The stored evaluation results can only become outdated when a variable or parameter is changed. Because of this, a process's condition list is cleared when that process executes a statement that may result in such a change. After the following occurrences, the conditions need to be updated:

- Receiving a send or broadcast message with parameters.
- Returning from a method call with return parameters.
- Granting a statement request.

A statement request can be followed by the execution of a data statement or a process statement with side effects. Both are able to change variables or parameters. The send, broadcast and delay statement do not belong to this list since the expressions of these statements are not allowed to have side effects and therefore can not change variables or parameters.

4.5.2 SmartGuards

SmartGuards are guards that take advantage of the absence of concurrency (parallelism) within a process. Without concurrency, the dependencies of a guard condition are unable to change. Such a non-concurrent guard can be evaluated even before it is placed in the process tree. If the condition holds, building the guard node can be skipped. In the other case, building the whole branch can be left out with the following results:

- One or more process nodes do not have to be built.
- One or more threads do not have to be created.
- One or more requests do not have to be scheduled.
- The scheduler does not have to test whether the branch's requests are executable or not.

This saves a considerable amount of computing time in spite of the sacrifices that must be made to evaluate the guard condition and to find out whether or not the guard is concurrent.

The abort and interrupt are the only statements that introduce concurrency within a process. This means that a guard is concurrent if (and only if) it is nested in an abort or interrupt statement. Concurrency can be determined by examining the process tree; a node is concurrent when the path from its parent to the root contains an interrupt or abort node.

Before it places itself into the process tree, a smartGuard always checks if it is concurrent. A concurrent smartGuard behaves itself just like a conventional guard. A non-concurrent smartGuard saves computing time by skipping unnecessary actions as stated above.

An exception can be made for a smartGuard nested in S_1 of a (S_1 *abort* S_2) statement since S_1 is discarded as soon as S_2 performs an execution step. The execution step may change the guard's dependencies. Nevertheless, the guard is discarded because of the abort action. As long as the guard exists, its dependencies will not change. Such a guard can therefore be treated as a non-concurrent guard. Another exception can be made for a smartGuard guarding

any statement in S_2 aside from the first. This is because the execution of the first statement discards S_1 so that the guard's dependencies can not change anymore.

5 Translation Example

5.1 Data Class Translation

To explain how a POOSL data class definition has to be translated into a C++ class we define a class *Complex*, which is used to model complex numbers.

```
data class Complex
instance variables
        re: Integer
        im: Integer

instance methods
init(r: Integer; i: Integer): Complex
        re := r;
        im := i;
        return(self).

real: Integer
        return(re).

imag: Integer
        return(im).

add(comp: Complex): Complex
        | i: Integer; r: Integer; res: Complex |
        r := self real + (comp real);
        i := self imag + (comp imag);
        res := new(Complex) init(r,i);
        return(res).
```

5.1.1 Data Class Definition

The C++ translation of this example begins with a definition of the *Complex_* class, which is derived from the *NonPrimitiveDataObject* class. The underscore "_" character is used to prevent conflicts with other identifiers.

```
1  class Complex_ :public NonPrimitiveDataObject
2  {
3    Complex_() :NonPrimitiveDataObject(2, "re", "im")    {  }
4    public:
5    static Complex_& New()        { return *(new Complex_);  }
6    const char *GetClassName()    { return "Complex_";       }
7    size_t      GetClassSize()    { return sizeof(Complex_); }
8
9    Complex_& init_(ExprArray);
10   Integer& real_();
11   Integer& imag_();
12   Complex_& add_(ExprArray);
13 };
```

Line 3 defines the constructor of the class, which is used for initialising every new class instance. The parameters that are supplied to the constructor of the *NonPrimitiveDataObject* class instruct it to create two instance variables, one named "re" and one named "im". They will be contained in a *VarArray* and are accessible through *self*.

Next, three member functions are defined that are the same for every data class (except for the class identifier). Function *New* is used for the translation of the POOSL expression *new*. It creates a new *Complex_* object on the heap. Function *GetClassName* is used to identify the class of an instance by its name, and *GetClassSize* is used for informing the deepCopy

procedure about the size of an instance. The class name is used for printing debugging information and the size is required for making a copy of the object.

Lines 9-12 define the prototypes of the C++ member functions that represent the data class methods. Some functions have an *ExprArray* as argument because they must be called with one or more input parameters. The member functions themselves are defined next.

5.1.2 Data Class Methods

```
15 Complex_& Complex_::init_(ExprArray i)
16 {
17   LocalArray l(1, self);
18   i.SetNames("r", "i");
19   self.Assign("re", ((Integer&) i["r"]));
20   self.Assign("im", ((Integer&) i["i"]));
21   return (Complex_&) Return(self);
22 }
```

Lines 15-22 show the translation of the *init* method. Expression *self* refers to the data object that is executing the method. The *LocalArray l* is used to contain all the method's local variables. It is possible that the *self* object is not referred by any variable or parameter (when executing *new(Complex) init(1,1)* for example). So, to prevent the object from being cleaned up by the garbage collector, it is assigned to an element in the *LocalArray*. After the function has returned, the *LocalArray* will be deconstructed which implies that the reference to the *self* object will be deleted.

The method's input parameters are passed by an *ExprArray*. At line 18, the names of the parameters are set by the *SetNames* function. Naming of parameters and variables is optional. Without names, the variables and parameters can be addressed by their array index number. With names, they can be addressed using strings. In addition, using names has the advantage that the C++ program can display information about the variables and parameters at run-time.

The [] operator as in lines 19-20 is used to access a variable in a *LocalArray* or parameter in an *ExprArray*. It can address it by index number or by name string. The return type of the operator is a *DataObject*, because it must be able to return all types of data objects. The returned object must therefore be cast to the appropriate type by using a typecast. The *Assign* function in line 19 assigns input parameter "r" to instance variable "re" (which is contained in *self*).

Line 21 shows how an expression (in this case *self*) has to be returned. The *Return* function marks the return object in such a way that the object will not be reclaimed by the garbage collector (by making the reference count negative). The mark will be cleared when the object is assigned to its receiver (a variable or parameter). This means that the object is temporarily protected against the garbage collector. This is necessary because the references to a return object may all be deleted when the *LocalArray* is deconstructed. Without the protection, the return object would then be reclaimed before it could be assigned to its receiver. When the object does not have a receiver, this must notified to the garbage collector by using the *Cleanup* function.

```
24 Integer& Complex_::real_()
25 {
26   LocalArray l(1, self);
27   return (Integer&) Return(((Integer&) self["re"]));
28 }

30 Integer& Complex_::imag_()
31 {
32   LocalArray l(1, self);
33   return (Integer&) Return(((Integer&) self["im"]));
34 }
```

Lines 24-34 show nothing new except for returning an instance variable.

```
36 Complex_& Complex_::add_(ExprArray i)
37 {
38   LocalArray l(4, self, "i", "r", "res");
39   i.SetNames("comp");
40   l.Assign("r", (self.real_() + ((Complex_&) i["comp"]).real_()));
41   l.Assign("i", (self.imag_() + ((Complex_&) i["comp"]).imag_()));
42   l.Assign("res", Complex_::New().init_(
43     ExprArray(2,
44       &((Integer&) l["r"]),
45       &((Integer&) l["i"])
46     )
47   ));
48   return (Complex_&) Return(((Complex_&) l["res"]));
49 }
```

The + operator in lines 40-41 represents a primitive method of the *Integer* class. Sending messages to objects in C++ is similar as in POOSL (except for the dot "." and parameters). Parameters must be passed by an *ExprArray* instance placed on the call stack as in lines 42-47 with the *init* message. Each element of the *ExprArray* instance refers to a parameter data object. These references are deleted when the *init* message call terminates because then the call stack is removed which implies deconstruction of the *ExprArray* instance.

5.2 Process Class Translation

The next example shows how a POOSL process class has to be translated to a C++ class. First, we define the *Receiver* class.

process class	*Receiver*(*myID*: *Integer*)
instance variables	count: *Integer*
communication channels	*in*
message interface	*in* ? *packet*(*Integer*)
	in ? *reset*
initial method call	*init*()()
instance methods	

init()()
```
    count:=0;
    loop()() abort (in ? reset; init()()).
```

loop()()
```
    | id: Integer |
    in ? packet(id | id = myID);
    count := count + 1;
    loop()().
```

5.2.1 Process Class Definition

The C++ translation starts with de definition of the *Receiver_* class, which is derived from the *Process* and *VarArray* class. The *VarArray* is used to contain all the instance variables of the *Receiver_* class.

```
1  class Receiver_ :public Process, public VarArray
2  {
3    public:
4    ProcessConnector *in_;
5
6    Receiver_(Cluster *, const char *);
7    void StartUp(ExprArray &);
8
9    ReturnObject *loop_(ProcessNode *);
10   Boolean      &loop_Cond_1_1(ExprArray &, RetArray &,
11                               VarArray &                    );
12   ReturnObject *init_(ProcessNode *);
13   Tail         *init_Block_2_1(ProcessNode *, ExprArray &,
14                                RetArray &, VarArray &      );
15   Tail         *init_Block_2_2(ProcessNode *, ExprArray &,
16                                RetArray &, VarArray &      );
17 };
```

Line 4 defines a pointer to a *ProcessConnector* object. This object will be created and connected to a channel at run-time. The identifier *in_* corresponds to the communication channel in the POOSL specification. Next, the prototype of the class constructor is defined, followed by the prototypes of the member functions. Each process class method is translated by one or more member functions. A condition in a conditional receive statement must be translated into a member function as shown in lines 10-11. This is necessary because a condition can not be evaluated beforehand. The scheduler may evaluate a condition at any time by executing its function member. Process statements that start up threads (such as select, interrupt and abort) are decomposed into multiple member functions. These member functions are called blocks and each block corresponds to a node in the process tree. The blocks will be executed in (pseudo) parallel by the threads. Lines 13-16 show two blocks that are used to implement an interrupt statement.

5.2.2 Process Class Constructor

```
19 Receiver_::Receiver_(Cluster *owner, const char *name)
20 : Process(owner, name),
21   VarArray(2, "myID", "count")
22 {
23   in_ = new ProcessConnector(this,"in");
24 }
```

Lines 19-24 show the definition of the class constructor. Parameter *owner* is used to inform each *Receiver_* instance about its owner (a cluster object) and *name* is for identification of an instance at run-time. The purpose of the *VarArray* constructor is to create a container for the instantiation parameter "myID" and instance variable "count". Line 23 shows the statement that creates a new *ProcessConnector* object named "in".

5.2.3 Process Start-up

```
26 void Receiver_::StartUp(ExprArray &i)
27 {
28   Assign(i.DeepCopy());
29   InitialMethodCall((ProcessMethod) &Receiver_::init_);
30 }
```

Every process class has a *StartUp* member function. Its purpose is to initiate the process by calling its initial method. The *ExprArray* is used pass the instantiation parameters.

According to the definition of the process part of POOSL [PV97], expression parameters of a cluster are syntactically substituted by the corresponding instantiation expressions. These expression parameters may be used in the instantiation expressions of sub-clusters and/or processes. Syntactical substitution ensures that each process gets its own private (not shared with other processes) set of instantiation parameters.

In the C++ implementation, the instantiation expressions are immediately evaluated and bound to the corresponding expression parameters. To make sure that instantiation parameters will not be shared, each process instance gets a deepCopy (see line 28). However, side effects in the instantiation expressions may cause problems since these effects can accumulate at subsequent evaluations. This is because all instantiation expressions work on the same set of instantiation parameters (deepCopies are made later at process start-up). To solve this problem, each expression parameter used in the instantiation expressions is substituted by its deepCopy.

The *Assign* function binds the deepCopy to the *VarArray* that is inherited by the *Receiver_* class. This way, all its member functions have access to the instantiation parameters. The statement at line 29 actually starts up the process instance by creating a thread that executes the initial method.

5.2.4 Process Methods

```
32 ReturnObject *Receiver_::loop_(ProcessNode *MyNode)
33 {
34   ReturnObject *ret = NULL;
35   VarArray &l = *(new VarArray(1, "id"));
36   MyNode->Push(&l);
37
38   MyNode->Receive(in_, "packet(1)",
39     new ParamArray(1,
40       l("id")
41     ),
42     new Condition(this,
43       (CondExpr) &Receiver_::loop_Cond_1_1,
44       NOEXPR,
45       NORET,
46       1
47     )
48   );
49
```

```
50    MyNode->Statement();
51    self.Assign("count",
52      (((Integer&) self["count"]) + Integer::New(1L))
53    );
54
55    MyNode->Statement();
56    ret = new Tail((ProcessMethod) &Receiver_::loop_,
57      NULL
58    );
59
60    delete MyNode->Pop();
61    return( ret );
62  }
```

Lines 32-62 show the member function that represents the *loop* method. The *MyNode* parameter always provides access to the process tree node which thread is currently executing the function. Variable *ret* is used for returning a *Tail* object, a *RetArray* object or *NULL*. The first case is for returning with tail recursion, the second is for returning with output parameters, and *NULL* is for returning without output parameters.

At line 35, a new *VarArray* object is created on the heap. This object is used to contain all the method's local variables (in this case *id*). Next, the *VarArray* is registered at the current process node by pushing it on the cleanup stack. The process node uses this cleanup stack to remove all objects from the heap that become dispensable when statements are aborted.

Lines 38-48 show the implementation of a conditional receive statement. A receive statement must (even as all other process statements) be requested so that the scheduler can choose to grant it. Requests are created and submitted by the current process node (*MyNode*). In this case, the request is a *ReceiveRequest* with the following contents:

- A pointer to the current process node.
- A pointer to *ProcessConnector in_*.
- The message name *packet(1)* (The number between brackets indicates the number of parameters).
- A *ParamArray* with the address of local variable *id*.
- A *Condition* with the address of the member function that represents the expression of the receive condition.

After submitting the request, the process node blocks its thread by waiting for a signal from the scheduler. The pointer to the current process node is used to identify the node that submitted the request. This way, the scheduler knows which node it must signal when granting the request.

The pointer to *in_* is used to designate the channel whereto the request must be sent. After sending, this pointer indicates where the request entered the channel hierarchy, which is used for matching communication partners (see paragraph 3.5.2).

The message name identifies the request by its message name and the number of parameters. A channel uses this string to find matching communication partners. Because the string has the number of parameters added to it, only messages with the same number of parameters will match.

When receiving a message with parameters, the parameters must be bound to their receivers (variables and/or parameters). In order to do so, the addresses of these receivers must be known by the receive request. This is why the *ParamArray* stores addresses to variables and/or parameters (in contrast to the *VarArray*, *ExprArray* and *RetArray*, which store addresses to data objects). The () operator is used to get the address of the *id* variable.

The *Condition* object supplies the request with the address of the member function that implements the condition expression. The expression must have a way to access the method's local and instance variables and the input and output parameters. This is why the method's environment (the variables and parameters) is stored in the *Condition* object. At evaluation of the expression, the environment is passed to the member function. With *l* the local variables are passed and with *this* the instance variables. *NOEXPR* and *NORET* indicate that there are no input and return parameters respectively. If there are no local variables, the *l* should be replaced with *NOVAR* to indicate the absence.

Lines 50-53 show nothing that we have not seen already except for the *Statement* function call. The call is used for requesting permission to execute the process statement that follows the call. The current process node (*MyNode*) sends the request (a *StatementRequest*) directly to the scheduler. After that, it blocks its thread by waiting for a signal from the scheduler.

Lines 55-58 show also the request of a process statement. In this case, the statement is a tail-recursive call of method *loop*. A new *Tail* object is created and assigned to *ReturnObject ret*. The object contains the address of the member function that represents method *loop*. A Tail object can also contain a new *ExprArray* object for passing parameters to the method. Method *loop* is called without parameters, which is indicated by *NULL*. The actual call will be executed after the current member function has returned. Before that, *VarArray l* must be removed and unregistered at the current process node. The statement at line 60 takes care of this by popping the *VarArray* from the cleanup stack and deleting it.

```
64 Boolean &Receiver_::loop_Cond_1_1(ExprArray &, RetArray &, VarArray &l)
65 {
66   return ( (((Integer&) l["id"]).IsEqual(((Integer&) self["myID"]))) );
67 }
```

Lines 64-67 show the member function that implements the expression of the receive condition. The arguments are used to pass the method's environment to the expression. Because the function is a member of the *Receiver_* class, it can access instance variables through *self*. Member function *IsEqual* is the C++ translation of primitive method =.

```
69 ReturnObject *Receiver_::init_(ProcessNode *MyNode)
70 {
71   ReturnObject *ret = NULL;
72
73   MyNode->Statement();
74   self.Assign("count", Integer::New(0L));
75
76   ret = MyNode->Abort(this,
77     NOEXPR,
78     NORET,
79     NOVAR,
80     (ProcessBlock) &Receiver_::init_Block_2_1,
81     (ProcessBlock) &Receiver_::init_Block_2_2
82   );
83
84   return( ret );
85 }
```

The implementation of the *init* method shows nothing new except for the translation of the abort statement at lines 76-82. The *Abort* function changes the current process node into an abort node and builds two children nodes. Each child's thread will execute a composition of statements. One child will execute the abortable composition and the other the aborting composition. Each composition is represented by a member function (see line 80 and 81). When necessary, the current method's variables and/or parameters must be passed to the compositions. In this case, the compositions use neither variables nor parameters, which explains the use of *NOEXPR*, *NORET* and *NOVAR*.

The *Abort* function returns when one of the abort node's children terminates. A child that wants to execute a tail-recursive call must terminate its thread by returning a *Tail* object. This *Tail* object will then be received by the *Abort* function through thread joining. When the thread has terminated, the *Abort* function transforms the abort node back into a leaf node and returns with the *Tail* object. Then, the *Tail* object is assigned to *ret* and returned to the caller of the current method. This caller will also execute the tail-recursive call.

```
87 Tail *Receiver_::init_Block_2_1(ProcessNode *MyNode, ExprArray &,
88                                  RetArray &, VarArray &)
89 {
90   Tail *ret = NULL;
91
92   MyNode->Statement();
93   MyNode->Call(this, (ProcessMethod) &Receiver_::loop_,
94     NULL,
95     NULL
96   );
97
98   return ( ret );
99 }
```

Lines 87-99 show the member function that represents the abortable composition of statements. Again, the arguments are used to pass the method's environment. At lines 92-96, we see the implementation of a recursive method call. The two *NULL* pointers indicate that the method has no input or output parameters respectively. The member function that represents the aborting composition follows below.

```
101 Tail *Receiver_::init_Block_2_2(ProcessNode *MyNode, ExprArray &,
102                                  RetArray &, VarArray &)
103 {
104   Tail *ret = NULL;
105
106   MyNode->Receive(in_, "reset(0)");
107
108   MyNode->Statement();
109   ret = new Tail((ProcessMethod) &Receiver_::init_,
110     NULL
111   );
112
113   return ( ret );
114 }
```

5.3 Cluster Class Translation

To explain how a POOSL cluster class has to be translated into a C++ class we define a class *Receiver_Cluster*, which describes two *Receiver* instances connected together by a channel.

cluster class	*Receiver_Cluster(clusID: Integer)*
communication channels	*in*
message interface	*in ? reset*(); *in ? packet(Integer)*
behaviour specification	(*Receiver(clustID* * 2)[*ch/in*] || *Receiver(clustID* *2 + 1)[*ch/in*])[*in/ch*]

5.3.1 Cluster Class Definition

The C++ translation of this example is the *Receiver_Cluster_* class, which is derived from the *Cluster* class.

```
1  class Receiver_Cluster_ :public Cluster
2  {
3    public:
4    Receiver_ *Receiver_1;
5    Receiver_ *Receiver_2;
6
7    ClusterConnector *in_;
8    Channel *in_i;
9
10   Receiver_Cluster_(Cluster *, const char *);
11   void StartUp(ExprArray &);
12 };
```

A cluster can contain process, sub-cluster, channel and/or connector instances. For every such instance, the cluster has a pointer in its class definition. When the cluster class is instantiated, its class constructor will create all the instances and assign them to the pointers. The constructor also connects the channels to the appropriate connectors. The constructors of sub-clusters call the constructors of their sub-clusters and so on. This way a complete system will be instantiated in the same order as if we were traversing the cluster tree structure in pre-order.

After constructing the complete system, the processes must be started up. This is the *StartUp* function's task. A cluster's *StartUp* function calls the *StartUp* function of every process and

sub-cluster of that cluster. The *StartUp* function of a process starts up the process by executing its initial method. If a cluster or process has instantiation parameters, they must be passed to its *StartUp* function by an *ExprArray*.

5.3.2 Cluster Class Constructor

```
14 Receiver_Cluster_::Receiver_Cluster_(Cluster *owner, const char *name)
15 : Cluster(owner, name)
16 {
17   Receiver_1 = new Receiver_(this,"Receiver_1");
18   Receiver_2 = new Receiver_(this,"Receiver_2");
19
20   in_  = new ClusterConnector(this,"in");
21   in_i = new Channel(this,"in");
22   in_i->Connect(in_);
23   in_i->Connect(Receiver_1->in_);
24   in_i->Connect(Receiver_2->in_);
25 }
```

Lines 14-25 show the definition of the class constructor. The statements at line 17-18 instantiate the two *Receiver_* objects. The next two statements instantiate the *Channel* and *ClusterConnector* object. All instances are given a string with their name so that they can identify themselves at run-time. Lines 22-24 show the statements that connect the channel according to the behaviour specification. First the channel is connected to the cluster connector and then to the process connectors of the two *Receiver_* instances.

5.3.3 Cluster Start-up

```
27 void Receiver_Cluster_::StartUp(ExprArray &i)
28 {
29   i.SetNames("clustID");
30
31   Receiver_1->StartUp(
32     *(new ExprArray(1,
33       &(((Integer&) i["clustID"].DeepCopy()) * Integer::New(2L))
34     ))
35   );
36
37   Receiver_2->StartUp(
38     *(new ExprArray(1,
39       &((((Integer&) i["clustID"].DeepCopy()) * Integer::New(2L)) +
40                                                 Integer::New(1L))
41     ))
42   );
43 }
```

Lines 27-43 show the cluster's *StartUp* function. *ExprArray i* is used to pass the cluster's instantiation parameters. The *StartUp* function starts up the two the *Receiver_* instances by calling their *StartUp* function. The cluster's *StartUp* function may have terminated even before the process threads are running. This is why the *ExprArray*'s are placed on the heap (with *new*). Because when placed on the stack, the *ExprArray* risks being removed before it has been passed to the thread. The expression parameter "clustID" used in the instantiation expressions is substituted by its deepCopy for the reason explained in paragraph 5.2.3.

6 From POOSL to Executable

6.1 From POOSL to C++

For automatically translating a POOSL specification into C++ sources, a POOSL compiler has been developed [Lei97]. The compiler runs in a Smalltalk [GR89] environment (on a Windows 95 platform).

As starting point, we take a complete POOSL specification written in a single ASCII file. Its grammar must comply with the concrete POOSL syntax [Lei97]. Furthermore, the specification must meet the context conditions. All context conditions need to be checked manually since the compiler is not (yet) capable of performing this task.

First, create a directory where the compiler can store the C++ source files. After that, start the compiler and supply it the ASCII file. The compiler will then prompt a dialog window with among others the following fields:

- System name
- Path
- Use variable names

The "System name" field needs to be supplied with the name of the executable that will be built by the C++ compiler. This name is case sensitive and may not be the same as any used POOSL class name (data, process or cluster).

The "Path" field refers to the destination directory where the POOSL compiler will store the C++ sources. The last character of this path must be a blackslash "\".

The "Use variable names" option can be enabled to let the executable use strings in stead of index array numbers to address variables and parameters. Using strings has the advantage that variables and parameters can be printed with their name at run-time, which is very useful for debugging purposes. It is also safer because the arrays that contain the variables and parameters will never be accessed outside their boundaries. However, the disadvantage of using strings is that the executable will run slightly slower, since each string must be looked up to find its corresponding array index number.

After filling in the fields, click the "Accept" button. The compiler will now generate the C++ sources and stores them in the destination directory. The next paragraph will explain how to build an executable program from these sources.

6.2 From C++ to Executable

The C++ sources generated by the POOSL compiler must be compiled and linked together with the POOSL library into an executable program. We will create this program on a UNIX platform (the HP UX 9.05 operating system), using the GNU C++ compiler and *make* utility.

The whole compilation and linking process is automated using makefiles. The makefiles rely on the directory structure shown in Figure 6.1. The makefile in directory *src* is used to build the POOSL library. The library will be stored in directory *lib*.

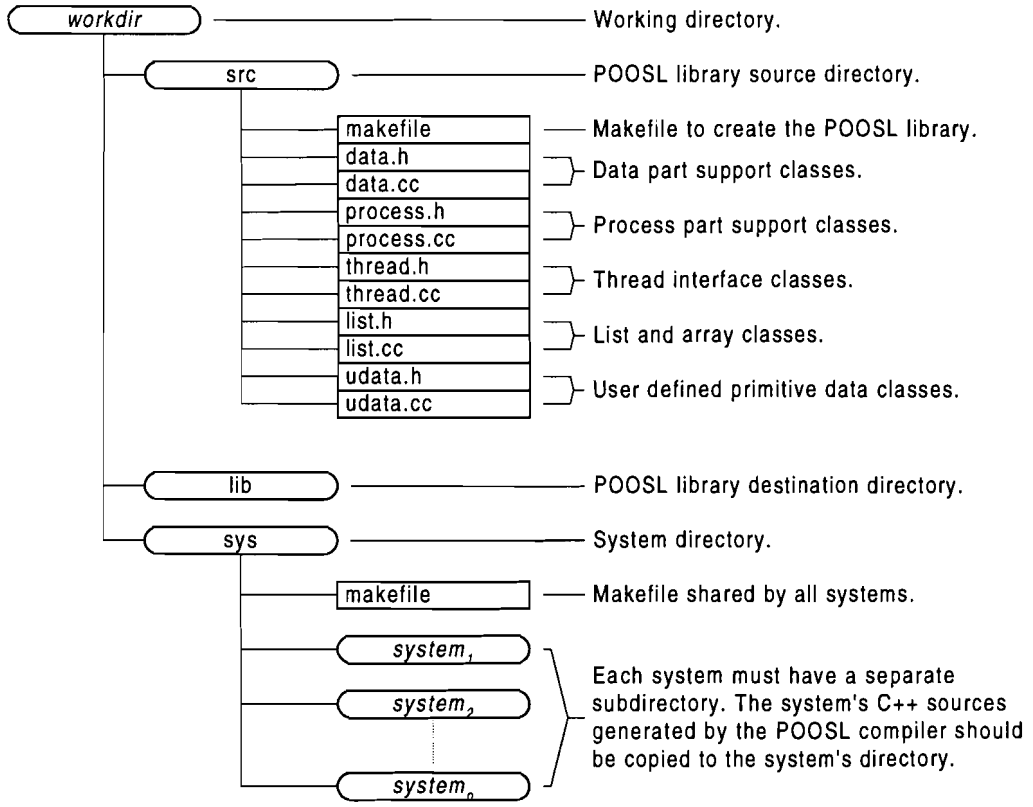


Figure 6.1: Directory Structure of *workdir*

Every complete POOSL specification is called a system. The C++ sources that are generated by compiling a system should be copied into a separate system directory. To compile the sources, all systems use the makefile that resides in the *sys* directory. Each system also has its own makefile. This makefile is generated by the POOSL compiler and must be copied to the system's directory just like the C++ sources. When *make* executes this makefile, it invokes the makefile in the *sys* directory that on its turn invokes the makefile in the *src* directory. Together, the makefiles will build all the necessary files and link them into an executable program.

The following steps show how the C++ sources of a system should be build into an executable program:

- Create a system directory in the *sys* directory. The system name can be used as the directory's name but another name is also possible.
- Copy all the files generated by the POOSL compiler (C++ sources and makefile) to the system directory.
- Change to the system directory.
- Except for the makefile, all files in de system directory should have capitalised filenames. To lower the case of the makefile's filename, move *MAKEFILE* to *makefile*.

- Build the executable by running *make*. By default, this utility will use the makefile located in the current directory. Be sure to use GNU's version of *make* since not all versions use the same makefile syntax. GNU's version is probably located in directory */usr/gnu/bin* so add this directory to the beginning of your searchpath. Your searchpath is correct when the command *make --version* displays GNU's version.

If all steps have been completed successfully, the system directory contains an executable program that represents the POOSL specification.

6.3 Run-time Information

For program debugging and verification, it can be very useful to let the program display information about its activities. The kind of information that will be displayed at run-time depends on the macros that are defined in the POOSL library sources. The macros are listed below together with their function.

- DBG_TREE Displays process tree information when the tree changes.
- DBG_REQ Displays requests sent to the scheduler or channel.
- DBG_GRANT Displays requests granted by the scheduler.
- DBG_INIT Displays initialisation of clusters, processes, channels and connectors.
- DBG_STATS *n* Displays statistical info such as number of scheduled and granted requests, rendezvous count, etcetera. The info is repeated every *n* granted non-delay requests.
- DBG_MSG Displays sent or received messages.
- DBG_FLOW *n* Displays a table of message flows between processes. The header of the table (a list of processes) is repeated after every *n* messages.

When a thread writes to the console, it can cause itself to block. If this thread is a process thread and all other threads are already blocked, this causes the scheduler thread to continue before its turn. The result is that the normal execution of the program gets disturbed. To prevent this problem, process threads should write to a file instead of the console. The console output of a program can be redirected to a file by adding " > *filename*" on the command line after the program's start-up command. When using only DBG_STATS and/or DBG_FLOW the program output does not have to be redirected because in this case the information is printed solely by the scheduler thread. If the scheduler thread blocks due to a console output, the next thread that continues is the scheduler itself since all other threads remain blocked until the scheduler grants a request. The execution of the program is therefore unaffected (except for waiting on the console).

7 Conclusions, Results and Future Work

7.1 Conclusions

In POOSL, data objects are created dynamically using the *new* statement. However, there exists no POOSL statement for deleting those objects. Data objects use memory and memory is a limited resource. Therefore, the C++ translation must use a garbage collection technique to reclaim the memory of data objects that have become dispensable.

An efficient implementation of the communication statements in combination with the select, abort and interrupt statements requires some kind of arbitration. This is because a process can not decide for itself with which process it is able to communicate, since this may depend on the decisions of other processes.

The interrupt and abort statements allow several statements to be active at the same time, each within its own local environment. This requires a context switching mechanism to switch execution of statements between the local environments. Context switching can be implemented in C++ by threads. A thread is a local environment wherein a sequence of statements can be executed. Context switching between threads can be controlled using semaphores.

By placing the guarded statement and each alternative of a select statement in its own local environment, we can easily implement nesting of statements using a divide and conquer strategy.

Each POOSL process executes its statements in parallel with other processes. This can be implemented in C++ by making a context switch between the processes after every executed process statement.

7.2 Results

A method has been developed for translating POOSL specifications into C++. The method covers complete translation of all POOSL statements included the POOSL extensions, namely the delay and broadcast statements. To offer the functionality that is required for translation, the POOSL library has been developed.

The translation method describes how to generate C++ sources from a POOSL specification. These sources compiled and linked together with the POOSL library result in an executable program that reflects the POOSL specification. The process of generating the C++ sources has been implemented in a POOSL to C++ compiler [Lei97].

Several case tests have been performed to test the translation method and the POOSL compiler as well. One of these tests was the PAR (Positive Acknowledgement with Retransmission) protocol (see [Voe95a] for the POOSL specification). Another test was a POOSL model of the elevator problem as described in [PV97]. All tests have been completed successfully.

7.3 Future Developments

7.3.1 Stacksize Estimation

Future versions of the POOSL library should have a way to parameterise the stacksize of each thread. This way, memory usage can be minimised. The POOSL to C++ compiler must then be equipped with heuristics to estimate the minimum required stacksize. Another solution is to use threads with the ability to enlarge the stacksize on the fly.

7.3.2 Performance Optimizations

The alternatives of the select statement do not necessarily have to be executed by separate threads. For each alternative, we can let the POOSL compiler determine the (nested) statement(s) that perform(s) the first execution step. The compiler must then generate the code that submits an appropriate request for every first execution step. All requests will be submitted at once without creating threads, which saves computing time.

7.3.3 Real-time Cyclic Garbage Collection

The reference counting garbage collector currently implemented in the POOSL library should be replaced by one that is able to reclaim cyclic garbage. This garbage collector should be suitable for real-time systems. The time that the scheduler must wait to synchronise POOSL time with the wall clock time (as explained in paragraph 3.6.2) can be exploited by performing garbage collection tasks.

7.3.4 Interfacing

User-primitive data classes can be developed to interface with external hardware but this will offer only polling techniques. We may want some kind of interrupt handler that is able to insert requests directly into a scheduler queue. Another possibility is to develop primitive process classes that handle an external interrupt by sending a certain message over a specific channel.

References

[Fel96] Felius, J.D. van
From POOSL to C++.
Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering, 1996.
Master's thesis.

[Gei96] Geilen, M.C.W.
Real-Time Concepts for Software/Hardware Engineering.
Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering, 1996.
Master's thesis.

[GR89] Goldberg, A and D. Robson
Smalltalk-80: The language.
Reading, Massachusetts: Addison-Wesley, 1989.

[Lei97] Leijer, M.L. de
POOSL-Compiler for Smalltalk and C++
Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering, 1997.
Master's thesis.

[OSF95] Open Software Foundation
OSF DCE Application Development Reference.
New Jersey: Prentice Hall, Vol. 2(1995).

[PS95] Plainfosse, D. and M. Shapiro
A Survey of Distributed Garbage Collection Techniques.
In: Proceedings Memory Management. International Workshop IWMM 95.
Kinross, UK, 27-29 Sept. 1995. Ed. by H.G. Baker.
Berlin, Germany: Springer-Verlag. 1995. p. 211-49.

[PVS95] Putten, P.H.A van der, J.P.M. Voeten and M.P.J. Stevens
Object-Oriented Co-Design for Hardware/Software Systems.
In: Proceedings of EuroMicro '95.
Como, Italy, 1995. Ed. by M. Cavanaugh.
Los Alamitos, California: IEEE Computer Society Press 1995, p. 718-726.

[PVS96] Putten, P.H.A van der, J.P.M. Voeten and M.P.J. Stevens
Behaviour-Preserving Transformations in SHE, A Formal Approach to Architecture Design.
In: Proceedings of EuroMicro '96.
1996, Ed. by P. Milligan and K. Kuchcinski.
Los Alamitos, California: IEEE Computer Society Press 1996, p. 12-27.

[PV97] Putten, P.H.A. van der and J.P.M. Voeten
Specification of Reactive Hardware/Software Systems.
Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering, 1997.
PhD Thesis.

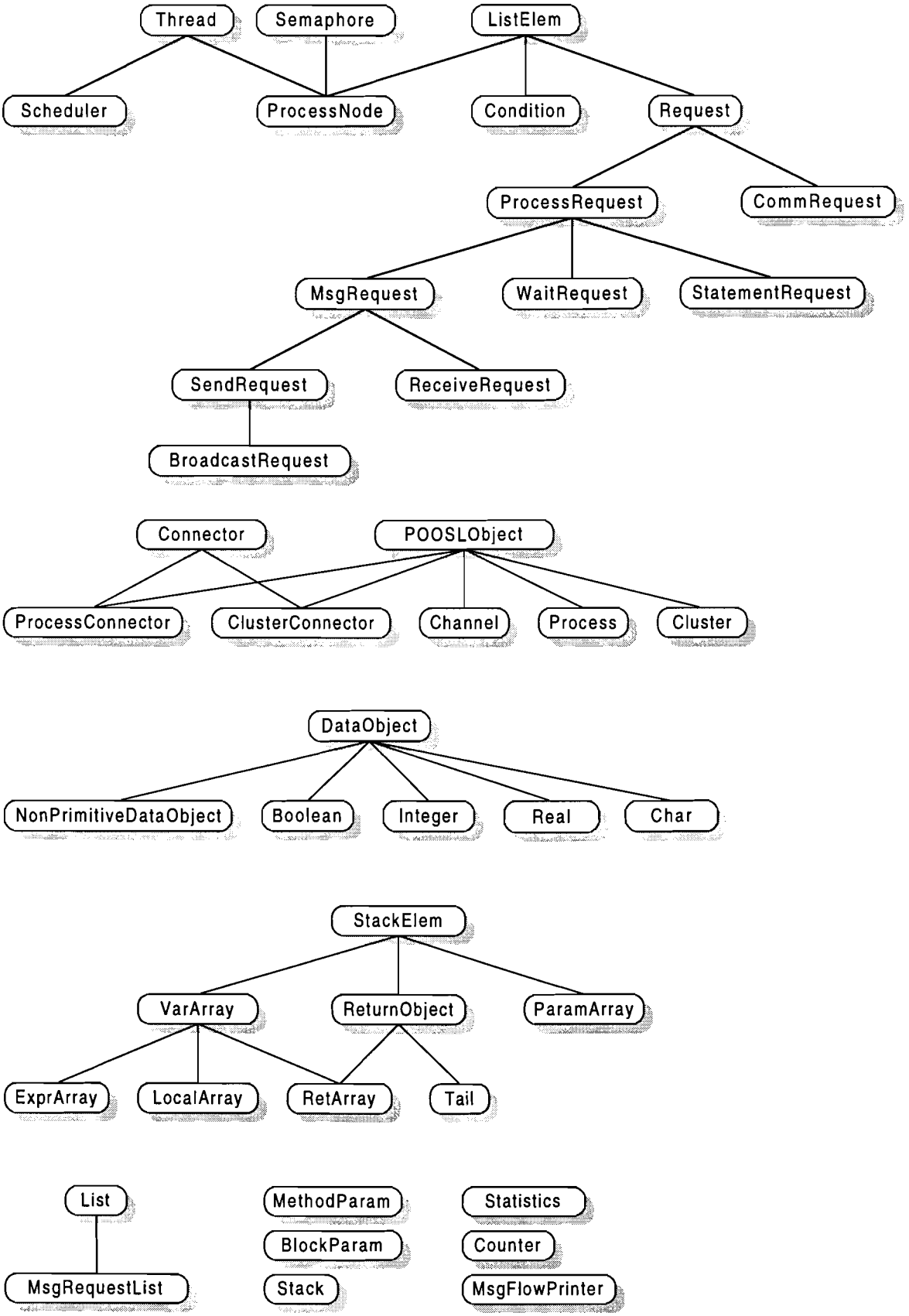
[Str92] Stroustroup B.
The C++ Programming Language.
Reading, Massachusetts: Addison-Wesley, 1992.

[Voe95a] Voeten, J.P.M.
POOSL: An Object-Oriented Specification Language for the Analysis and Design of Hardware/Software Systems.
Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering, 1995.
EUT Report 95-E-290.

[Voe95b] Voeten, J.P.M.
Semantics of POOSL: An Object-Oriented Specification Language for the Analysis and Design of Hardware/Software Systems.
Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering, 1995.
EUT Report 95-E-293.

[Wil92] Wilson, P.R.
Uniprocessor Garbage Collection Techniques.
In: Proceedings Memory Management. International Workshop IWMM 92. St. Malo, France. 17-19 Sept. 1992. Ed. by Y. Bekkers and J. Cohen.
Berlin, Germany: Springer-Verlag, 1992, p. 1-42.

Appendix A: POOSL Library Class Inheritance Structure



Appendix B: POOSL to C++ Translation

The translation function **P2CPP(SSpec)** represents the function that translates a POOSL system specification into C++. The **bold** text stands for the substitution of the corresponding syntax definition or translation function. Text printed in *italic* stands for the substitution of the corresponding POOSL identifier.

POOSL variable and parameter identifiers are translated into strings in the C++ code. This enables the C++ implementation to print debugging information during program execution. To increase run-time performance, the strings used in data expressions (**E**) may be replaced by ID numbers. The ID number of a variable/parameter is its position number (starting at position 0) in the corresponding variable/parameter list. Instance variables and instantiation parameters must be in the same list (of instance variables) starting with the instantiation parameters.

So, to disable variable/parameter debugging information and increase run-time performance, apply the following to the POOSL to C++ mapping:

- Replace all the strings in solid lined boxes by the corresponding identifier ID numbers.
- Leave out the code in dotted lined boxes.

Blocks are used to indicate a composition of process statements. A condition indicates a Boolean expression used for a receive or guard condition. Each block or condition is translated into a separate C++ member function. Blocks can be nested in select, abort and interrupt statements. A block or condition is designated by a variable number of indices. The first index indicates sequence number of the statement at nesting level 1 (the base level) where the block or condition is nested in. The second index indicates its sequence number at nesting level 2, and so on. This way it is always possible to find the indices of a block or condition when its location in the POOSL specification is known.

Syntax Definitions

SSpec ::= { **BSpec**, **Sys**^{d}, **Sys**^{p}, **Sys**^{c} }

Sys^{d} ::= **classDef**$^{d}_{1}$ ··· **classDef**$^{d}_{n}$
Sys^{p} ::= **classDef**$^{p}_{1}$ ··· **classDef**$^{p}_{m}$
Sys^{c} ::= **classDef**$^{c}_{1}$ ··· **classDef**$^{c}_{k}$

classDef^{d} ::=
data class *dataClass*
instance variables $instVar_1:ivType_1;\cdots;instVar_n:ivType_n$
instance methods **methodDef**$^{d}_{1}$···**methodDef**$^{d}_{m}$

methodDef^{d} ::=
$method(inputParam_1:iType_1,\cdots,inputParam_p:iType_p):mType$
$|localVar_1:lType_1;\cdots;localVar_k:lType_k|$
statement^{d}

classDef^{p} ::=
process class $processClass(instParam_1:ipType_1,\cdots,instParam_u:ipType_u)$
instance variables $instVar_1:ivType_1;\cdots;instVar_v:ivType_v$
communication channels $ch_1,\cdots,ch_c$
message interface $l_1;\cdots;l_l$
initial method call $initialmethod(\mathbf{E}_1,\cdots,\mathbf{E}_n)()$
instance methods **methodDef**$^{p}_{1}$···**methodDef**$^{p}_{m}$

methodDef^{p} ::=
$method(inputParam_1:iType_1,\cdots,inputParam_p:iType_p)$
$(retParam_1:rType_1,\cdots,retParam_q:rType_q)$
$|localVar_1:lType_1;\cdots;localVar_k:lType_k|$
block.

block$_{s1,\_,sj}$ ::= **statement**$^{p}_{s1,\_,sj,1}$;···;**statement**$^{p}_{s1,\_,sj,n}$

statement$^{p}_{s1,\_,sj}$ ::=
statement^{d}
| $ch!msg(\mathbf{E}_1,\cdots,\mathbf{E}_n)$
| $ch!*msg(\mathbf{E}_1,\cdots,\mathbf{E}_n)$
| $ch?msg(\mathbf{P}_1,\cdots,\mathbf{P}_m)$
| $ch?msg(\mathbf{P}_1,\cdots,\mathbf{P}_m|$**condition**$_{s1,\_,sj})$
| $method(\mathbf{E}_1,\cdots,\mathbf{E}_n)(\mathbf{P}_1,\cdots,\mathbf{P}_m)$
| sel **block**$_{s1,\_,sj,1}$ or ··· or **block**$_{s1,\_,sj,n}$ les
| [**condition**$_{s1,\_,sj}$] **block**$_{s1,\_,sj}$
| if **E** then **block**$_{s1,\_,sj,1}$ else **block**$_{s1,\_,sj,2}$ fi
| if **E** then **block**$_{s1,\_,sj}$ fi
| while **E** do **block**$_{s1,\_,sj}$ od
| **block**$_{s1,\_,sj,1}$ abort **block**$_{s1,\_,sj,2}$
| **block**$_{s1,\_,sj,1}$ interrupt **block**$_{s1,\_,sj,2}$
| delay **E**
| skip

P ::=
instVar
| *inputParam*
| *retParam*
| *localVar*
| *instParam*

condition$_{s1,\_,sj}$ ::= **E**

classDef^{c} ::=
cluster class $clusterClass(exprParam_1:epType_1,\cdots,exprParam_u:epType_u)$
communication channels $ch_1,\cdots,ch_c$
message interface $l_1;\cdots;l_l$
behaviour specification **BSpec**

```
BSpec          ::=  class^p(E_1,···,E_n)
                  |  class^c(E_1,···,E_n)
                  |  BSpec_1 || BSpec_2
                  |  BSpec \ {hiding_1,···,hiding_n}
                  |  BSpec [newLabel_1/oldLabel_1,···,newLabel_n/oldLabel_n]

statement^d    ::=  instVar    := E
                  |  localVar   := E
                  |  inputParam := E
                  |  retParam   := E
                  |  instParam  := E
                  |  statement^d_1;statement^d_2
                  |  E
                  |  while E do statement^d od
                  |  if E then statement^d fi
                  |  if E then statement^d_1 else statement^d_2 fi
                  |  return(E)

E              ::=  instVar
                  |  localVar
                  |  inputParam
                  |  retParam
                  |  instParam
                  |  exprParam
                  |  new (dataClass)
                  |  self
                  |  E method(E_1,···,E_n)
                  |  E_1 binaryOperator E_2
                  |  unaryOperator E
                  |  true
                  |  false
                  |  bunk
                  |  constInteger
                  |  iunk
                  |  constReal
                  |  runk
                  |  constCharacter
                  |  cunk
                  |  constString
                  |  nil
```

Definitions Required for the Translation Function

$e, e_1, \cdots, e_n \in \{ block_{s1,\_,sj} \mid j, s_1, \cdots, s_j \in N \} \cup \{ condition_{s1,\_,sj} \mid j, s_1, \cdots, s_j \in N \}$
$\langle e_1, \cdots, e_n \rangle \in List$

$\langle e_1, \cdots, e_n \rangle + \langle e_{n+1}, \cdots, e_{n+m} \rangle = \langle e_1, \cdots, e_n, e_{n+1}, \cdots, e_{n+m} \rangle$
$getFirst(\langle e, e_1, \cdots, e_n \rangle) = e$
$discardFirst(\langle e, e_1, \cdots, e_n \rangle) = \langle e_1, \cdots, e_n \rangle$
$| \langle e_1, \cdots, e_n \rangle | = n$

$LIST(block_{s1,\_,sj}) ::= \langle block_{s1,\_,sj} \rangle + SCAN(block_{s1,\_,sj})$

$SCAN(block_{s1,\_,sj}) ::= LIST(statement^{p}_{s1,\_,sj,1}) + \cdots + LIST(statement^{p}_{s1,\_,sj,n})$

$PROTOTYPE(list) ::=$
- PROTOTYPE(getFirst(list)) PROTOTYPE(discardFirst(list)) ,if | list | > 0
- ,otherwise

$P2CPP(list) ::=$
- P2CPP(getFirst(list)) P2CPP(discardFirst(list)) ,if | list | > 0
- ,otherwise

$IF(cond, thenText, elseText) ::=$
- thenText ,if cond is true
- elseText ,otherwise

$IF(cond, thenText) ::=$ IF(cond,thenText,)

$INPUTPARAM(x) ::=$
- true ,if expression/block/condition x refers to an *inputParam*
- false ,otherwise

$RETPARAM(x) ::=$
- true ,if expression/block/condition x refers to a *retParam*
- false ,otherwise

$LOCALVAR(x) ::=$
- true ,if expression/block/condition x refers to a *localVar*
- false ,otherwise

CONST(*constInteger***) ::= C++ representation of** *constInteger* **(a long)**
CONST(*constReal***) ::= C++ representation of** *constReal* **(a double)**
CONST(*constChar***) ::= C++ representation of** *constChar* **(a char)**
CONST(*constString***) ::= C++ representation of** *constString* **(a const char *)**

Note that the C++ class names of POOSL primitive data classes are: Boolean, Integer, Real, and Char. The string is a user primitive data class String_.

Note that the conversion by **CONST()** applies constraints to the value range of POOSL constants.
Note that *system* is the name of the system.
Note that the C++ identifiers that represent POOSL identifiers have an underscore (_) appended to their original name. This is done to prevent conflicts with C++ keywords and library functions.

Note that *unaryOperator* and *binaryOperator* stand for the textual representation of primitive data class operators. All operators are translated literally, except for $\mathbf{E}_1$ = $\mathbf{E}_2$ which has to be translated into (**P2CPP(**$\mathbf{E}_1$**)**.IsEqual(**P2CPP(**$\mathbf{E}_2$**)**)).
The behaviour specification (BSpec) of a cluster, must be broken down into instances of clusters, processes and channels and their interconnection. Because this is very complex to specify formally, the following "magic functions" are used:

ID(*class***)** **::=** **unique identification number for every instance of cluster or process class** *class*
ID **::=** **identifier of the corresponding cluster or process instance**
CHID() **::=** **unique channel identifier ()**
CHID **::=** **identifier of the corresponding channel instance**

Translation Function

```
P2CPP(SSpec)          ::=  P2CPP(Sys^d)
                         +  P2CPP(Sys^p)
                         +  P2CPP(Sys^c)
                         +  P2CPP_SYS(SSpec)

P2CPP(Sys^d)          ::=  P2CPP(classDef^d_1)
                         +  ...
                         +  P2CPP(classDef^d_n)

P2CPP(Sys^p)          ::=  P2CPP(classDef^p_1)
                         +  ...
                         +  P2CPP(classDef^p_m)

P2CPP(Sys^c)          ::=  P2CPP(classDef^c_1)
                         +  ...
                         +  P2CPP(classDef^c_k)

P2CPP(classDef^d)     ::=  HEADERFILE(classDef^d)
                         +  CPPFILE(classDef^d)

P2CPP(classDef^p)     ::=  HEADERFILE(classDef^p)
                         +  CPPFILE(classDef^p)

P2CPP(classDef^c)     ::=  HEADERFILE(classDef^c)
                         +  CPPFILE(classDef^c)

P2CPP_SYS(SSpec)      ::=  HEADERFILE(SSpec)
                         +  CPPFILE(SSpec)
```

HEADERFILE(classDef^d)	**File *dataClass*.h**

```
#ifndef __dataClass__
#define __dataClass__

#include "system.h"

class dataClass_ :public NonPrimitiveDataObject
{
  dataClass_() :NonPrimitiveDataObject(n,"instVar_1",···,"instVar_n") { }
  public:
  static dataClass_& New()    { return *(new dataClass_);  }
  const char *GetClassName() { return "dataClass";        }
  size_t      GetClassSize() { return sizeof(dataClass_); }
  PROTOTYPE(methodDef^d_1)
  ...
  PROTOTYPE(methodDef^d_m)
};

#endif
```

CPPFILE(classDef^{d})	File *dataClass*.cc

```
#include "system.h"

P2CPP(methodDef^d_1)
...
P2CPP(methodDef^d_m)
```

HEADERFILE(classDef^{p})	File *processClass*.h

```
#ifndef __processClass__
#define __processClass__

#include "system.h"

class processClass_ :public ProcessIF(v+u > 0,, public VarArray)
{
  public:
  ProcessConnector *ch1_;
  ...;
  ProcessConnector *chc_;
  processClass_(Cluster *, const char *);
  void StartUp(IF(n > 0,ExprArray &));
  PROTOTYPE(methodDef^p_1)
  ...
  PROTOTYPE(methodDef^p_m)
};

#endif
```

CPPFILE(classDef^{p})	File *processClass*.cc

```
#include "system.h"

processClass_::processClass_(Cluster *owner, const char *name)
: Process(owner,name)IF(v+u > 0,,
  VarArray(v+u,"instParam1",...,"instParamu","instVar1",...,"instVarv"))
{
  ch1_= new ProcessConnector(this,"ch1");
  ...;
  chc_= new ProcessConnector(this,"chc");
}

void processClass_::StartUp(IF(u > 0,ExprArray &i))
{
  IF(u > 0,Assign(i.DeepCopy());)
  InitialMethodCall((ProcessMethod)&processClass_::initialmethod_IF(n > 0,,
    new ExprArray(n,
      &P2CPP(E1),
      ...,
      &P2CPP(En)
    )));
}

P2CPP(methodDef^p_1)
...
P2CPP(methodDef^p_m)
```

HEADERFILE(classDef^c) | **File *clusterClass*.h**

```
#ifndef __clusterClass__
#define __clusterClass__

#include "system.h"

class clusterClass_ :public Cluster
{
  public:
  ClusterConnector *ch1_;
  ...;
  ClusterConnector *chc_;
  Channel *CHID();          }
  ...;                      }  all the channels in this cluster
  Channel *CHID();          }
  PROTOTYPE(BSpec)
  clusterClass_(Cluster *, const char *);
  void StartUp(IF(u > 0,ExprArray &));
};

#endif
```

CPPFILE(classDef^c) | **File *clusterClass*.cc**

```
#include "system.h"

clusterClass_::clusterClass_(Cluster *owner, const char *name)
: Cluster(owner,name)
{
  ch1_ = new ClusterConnector(this,"ch1");
  ...;
  chc_ = new ClusterConnector(this,"chc");
  CHID = new Channel(this,"CHID");          }
  ...;                                      }  all the channels in this cluster
  CHID = new Channel(this,"CHID");          }
  CONSTRUCT(BSpec)
  CONNECT(BSpec)
}

clusterClass_::StartUp(IF(u > 0,ExprArray &i))
{
  IF(u > 0,i.SetNames("exprParam1",...,"exprParamL");)
  STARTUP(BSpec)
}
```

HEADERFILE(SSpec)	File *system*.h

```
#ifndef __system__
#define __system__

  class dataClass1_;
  ...;                                  } for every data class in Sys^d
  class dataClassn_;

  class processClass1_;
  ...;                                  } for every process class in Sys^p
  class processClassm_;

  class clusterClass1_;
  ...;                                  } for every cluster class in Sys^c
  class clusterClassk_;


  #include "process.h"


  #include "dataClass1"
  ...;                                  } for every data class in Sys^d
  #include "dataClassn"

  #include "processClass1"
  ...;                                  } for every process class in Sys^p
  #include "processClassm"

  #include "clusterClass1"
  ...;                                  } for every cluster class in Sys^c
  #include "clusterClassk"

#endif
```

CPPFILE(SSpec) | **File *system*.cc**

```
#include "system.h"

class System :public Cluster
{
  public:
  Channel *CHID();                          } for all the channels at the top level of this system
  ...;                                      }
  Channel *CHID();                          }
  PROTOTYPE(BSpec)
  System(const char *);
  void StartUp();
};


System::System(const char *name)
: Cluster(NULL,name)
{
  CHID = new Channel(this,"CHID");          } for all the channels at the top level of this system
  ...;                                      }
  CHID = new Channel(this,"CHID");          }
  CONSTRUCT(BSpec)
  CONNECT(BSpec)
}

System::StartUp()
{
  STARTUP(BSpec)
}

int main(void)
{
  System Sys("system");
  Sys.StartUp();
  scheduler.Run();
  return ( 0 );
}
```

PROTOTYPE(methodDef^{d})

```
mType_& method_(IF(p > 0,ExprArray));
```

P2CPP(methodDef^{d})

```
mType_& dataClass_::method_(IF(p > 0,ExprArray i))
{
  LocalArray l(k + 1,self,"localVar1",...,"localVark");
  IF(p > 0,i.SetNames("inputParam1",...,"inputParamp");)
  P2CPP(statement^d);
}
```

PROTOTYPE(methodDef^{p})

```
ReturnObject *method_(ProcessNode *IF(p > 0,, ExprArray &));

list := LIST(block)
PROTOTYPE(list)
```

P2CPP(methodDef^{p})

```
list := LIST(block)

ReturnObject *processClass_::method_(ProcessNode *MyNodeIF(p > 0,, ExprArray &i))
{
  ReturnObject *ret = NULL;
  IF(k > 0,VarArray &l = *(new VarArray(k,"localVar1",···,"localVark"));)
  IF(q > 0,RetArray &r = *(new RetArray(q,"retParam1",···,"retParamq"));)
  IF(p > 0,i.SetNames("inputParam1",···,"inputParamp");)
  IF(k > 0,MyNode->Push(&l);)
  IF(q > 0,MyNode->Push(&r);)

  P2CPP(getFirst(list))

  IF(q > 0,ret = &r;)
  IF(q > 0,MyNode->Pop();)
  IF(k > 0,delete MyNode->Pop();)
  return ( ret );
}

P2CPP(discardFirst(list))
```

BSpec	**PROTOTYPE(BSpec)**
class^{p}(**E**$_1$,···,**E**$_n$)	*class*^{p}_ *****class*^{p}**_ID(***class*^{p}**);**
class^{c}(**E**$_1$,···,**E**$_n$)	*class*^{c}_ *****class*^{c}**_ID(***class*^{c}**);**
BSpec$_1$ || **BSpec**$_2$	**PROTOTYPE(Bspec**$_1$**)** **PROTOTYPE(Bspec**$_2$**)**
BSpec \ {*hiding*$_1$,···,*hiding*$_n$}	**PROTOTYPE(Bspec)**
BSpec[*newLabel*$_1$/*oldLabel*$_1$, ···, *newLabel*$_n$/*oldLabel*$_n$]	**PROTOTYPE(Bspec)**

BSpec	**CONSTRUCT(BSpec)**
class^{p}(**E**$_1$,···,**E**$_n$)	*class*^{p}**_ID** = new *class*^{p}_(this,"*class*^{p}**_ID**");
class^{c}(**E**$_1$,···,**E**$_n$)	*class*^{c}**_ID** = new *class*^{c}_(this,"*class*^{c}**_ID**");
BSpec$_1$ || **BSpec**$_2$	**CONSTRUCT(Bspec**$_1$**)** **CONSTRUCT(Bspec**$_2$**)**
BSpec \ {*hiding*$_1$,···,*hiding*$_n$}	**CONSTRUCT(Bspec)**
BSpec[*newLabel*$_1$/*oldLabel*$_1$, ···, *newLabel*$_n$/*oldLabel*$_n$]	**CONSTRUCT(Bspec)**

Bspec	CONNECT(BSpec)
channel *ch* **must be connected to processconnector** *pc* **of process instance** $class^{p}$**_ID**	*ch*->Connect(*class*^{p}**_ID**->*pc*);
channel *ch* **must be connected to clusterconnector** *cc* **of cluster instance** $class^{c}$**_ID**	*ch*->Connect(*class*^{c}**_ID**->*cc*);
channel *ch* **must be connected to clusterconnector** *cc* **of this cluster**	*ch*->Connect(*cc*);

BSpec	STARTUP(BSpec)
$class^{p}$(**E**$_1$,···,**E**$_n$)	*class*^{p}**_ID**->StartUp(**IF(n > 0,** *(new ExprArray(**n**, **&P2CPP(E**$_1$**)**, ···, **&P2CPP(E**$_n$**)**))));
$class^{c}$(**E**$_1$,···,**E**$_n$)	*class*^{c}**_ID**->StartUp(**IF(n > 0,** *(new ExprArray(**n**, **&P2CPP(E**$_1$**)**, ···, **&P2CPP(E**$_n$**)**))));
BSpec$_1$ || **BSpec**$_2$	**STARTUP(BSpec**$_1$**)** **STARTUP(BSpec**$_2$**)**
BSpec \ {*hiding*$_1$,···,*hiding*$_n$}	**STARTUP(BSpec)**
BSpec[*newLabel*$_1$/*oldLabel*$_1$, ···, *newLabel*$_n$/*oldLabel*$_n$]	**STARTUP(BSpec)**

statement^d	**P2CPP(statement^d)**
instVar := **E**	self.Assign("*instVar*", **P2CPP(E)**)
localVar := **E**	l.Assign("*localVar*", **P2CPP(E)**)
inputParam := **E**	i.Assign("*inputParam*", **P2CPP(E)**)
retParam := **E**	r.Assign("*retParam*", **P2CPP(E)**)
instParam := **E**	self.Assign("*instParam*", **P2CPP(E)**)
statement$^d_1$;statement$^d_2$	**P2CPP(statement$^d_1$);** **P2CPP(statement$^d_2$)**
E	**P2CPP(E)**.Cleanup()

while **E** do
  statement^d
od
	while (**P2CPP(E)**.IsTrue())
	{
	  P2CPP(statement^d);
	}

if **E** then
  statement^d
fi
	if (**P2CPP(E)**.IsTrue())
	{
	  P2CPP(statement^d);
	}

if **E** then
  statement$^d_1$
else
  statement$^d_2$
fi
	if (**P2CPP(E)**.IsTrue())
	{
	  P2CPP(statement$^d_1$);
	}
	else
	{
	  P2CPP(statement$^d_2$);
	}

return(**E**)	return (*mType_&*) Return(**P2CPP(E)**)

statement$^{p}_{s1,\_,sj}$	**LIST(statement**$^{p}_{s1,\_,sj}$**)**
statement^{d}	⟨⟩
ch!*msg*(**E**$_1$, ··· ,**E**$_n$)	⟨⟩
ch!*msg*(**E**$_1$, ··· ,**E**$_n$)	⟨⟩
ch?*msg*(**P**$_1$, ··· ,**P**$_m$)	⟨⟩
ch?*msg*(**P**$_1$, ··· ,**P**$_m$ |**condition**$_{s1,\_,sj}$)	⟨**condition**$_{s1,\_,sj}$⟩
method(**E**$_1$, ··· ,**E**$_n$)(**P**$_1$, ··· ,**P**$_m$)	⟨⟩
sel **block**$_{s1,\_,sj,1}$ or . . . or **block**$_{s1,\_,sj,n}$ les	**LIST(block**$_{s1,\_,sj,1}$**)** + ··· + **LIST(block**$_{s1,\_,sj,n}$**)**
[**condition**$_{s1,\_,sj}$] **block**$_{s1,\_,sj}$	⟨**condition**$_{s1,\_,sj}$⟩ + **LIST(block**$_{s1,\_,sj}$**)**

if **E** then
block$_{s1,\_,sj,1}$
else
block$_{s1,\_,sj,2}$
fi
	SCAN(block$_{s1,\_,sj,1}$**)** + **SCAN(block**$_{s1,\_,sj,2}$**)**

if **E** then **block**$_{s1,\_,sj}$ fi	**SCAN(block**$_{s1,\_,sj}$**)**
while **E** do **block**$_{s1,\_,sj}$ od	**SCAN(block**$_{s1,\_,sj}$**)**
Block$_{s1,\_,sj,1}$ abort **block**$_{s1,\_,sj,2}$	**LIST(block**$_{s1,\_,sj,1}$**)** + **LIST(block**$_{s1,\_,sj,2}$**)**
Block$_{s1,\_,sj,1}$ interrupt **block**$_{s1,\_,sj,2}$	**LIST(block**$_{s1,\_,sj,1}$**)** + **LIST(block**$_{s1,\_,sj,2}$**)**
Delay **E**	⟨⟩

PROTOTYPE(block$_{s1,_,sj}$)

```
Tail    *method_Block_s1_…_sj(ProcessNode *, ExprArray &, RetArray &, VarArray &);
```

PROTOTYPE(condition$_{s1,_,sj}$)

```
Boolean &method_Cond_s1_…_sj(ExprArray &, RetArray &, VarArray &);
```

P2CPP(condition$_{s1,_,sj}$)

```
Boolean &processClass_::method_Cond_s1_…_sj(ExprArray &IF(INPUTPARAM(E),i), RetArray
&IF(RETPARAM(E),r), VarArray &IF(LOCALVAR(E),l))
{
  return ( P2CPP(E) );
}
```

P2CPP(block$_{s1,_,sj}$)

```
Tail *processClass_::method_Block_s1_…_sj(ProcessNode *MyNode,
  ExprArray &IF(INPUTPARAM(block_{s1,_,sj}),i),
  RetArray  &IF(RETPARAM(block_{s1,_,sj}),r),
  VarArray  &IF(LOCALVAR(block_{s1,_,sj}),l))
{
  Tail *ret = NULL;
  P2CPP(statement^p_{s1,_,sj,1})
  ...
  P2CPP(statement^p_{s1,_,sj,n})
  return ( ret );
}
```

E	**P2CPP(E)**
instVar	((*ivType_*&) self["*instVar*"])
localVar	((*lType_*&) l["*localVar*"])
inputParam	((*iType_*&) i["*inputParam*"])
retParam	((*rType_*&) r["*retParam*"])
instParam	((*ipType_*&) self["*instParam*"])
exprParam	((*epType_*&) i["*exprParam*"].DeepCopy())
new(*dataClass*)	*dataClass_*::New()
self	self
E *method*(**E**$_1$,···,**E**$_n$)	**P2CPP(E)**.*method_*(**IF(n > 0,** ExprArray(**n**, **&P2CPP(E**$_1$**)**, ···, **&P2CPP(E**$_n$**)**)))
E$_1$ *binaryOperator* **E**$_2$	(**P2CPP(E**$_1$**)** *binaryOperator* **P2CPP(E**$_2$**)**)
unaryOperator **E**$_1$	(*unaryOperator* **P2CPP(E**$_1$**)**)
true	Boolean::New(TRUE)
false	Boolean::New(FALSE)
bunk	Boolean::Unk()
constInteger	Integer::New(**CONST(***constInteger***)**)
iunk	Integer::Unk()
constReal	Real::New(**CONST(***constReal***)**)
runk	Real::Unk()
constChar	Char::New(**CONST(***constChar***)**)
cunk	Char::Unk()
constString	String_::New(**CONST(***constString***)**)
nil	DataObject::Nil()

statement$^{p}_{s1,\_,sj}$	**P2CPP(statement**$^{p}_{s1,\_,sj}$**)**
statement^{d}	MyNode->Statement(); **P2CPP(statement**^{d}**);**

ch!*msg*(**E**$_1$,···,**E**$_n$)	MyNode->Send(*ch_*,"*msg*(**n**)"**IF(n > 0,**,
	  new ExprArray(**n**,
	  &P2CPP(E$_1$**)**,
	  ···,
	  &P2CPP(E$_n$**)**
	  )
	));

ch!*msg*(**E**$_1$,···,**E**$_n$)	MyNode->Broadcast(*ch_*,"*msg*(**n**)"**IF(n > 0,**,
	  new ExprArray(**n**,
	  &P2CPP(E$_1$**)**,
	  ···,
	  &P2CPP(E$_n$**)**
	  )
	));

ch?*msg*(**P**$_1$,···,**P**$_m$)	MyNode->Receive(*ch_*,"*msg*(**m**)"**IF(m > 0,**,
	  new ParamArray(**m**,
	  P2CPP(P$_1$**)**,
	  ···,
	  P2CPP(P$_m$**)**
	  )
	));

ch?*msg*(**P**$_1$,···,**P**$_m$ |**condition**$_{s1,\_,sj}$)	MyNode->Receive(*ch_*,"*msg*(**m**)",
	  IF(m > 0,new ParamArray(**m**,
	  P2CPP(P$_1$**)**,
	  ···,
	  P2CPP(P$_m$**)**
	  ),NULL),
	  new Condition(this,
	  (CondExpr)&*processClass_*::*method*_Cond_**s**$_1$**_**···**_s**$_j$,
	  IF(INPUTPARAM(condition$_{s1,\_,sj}$**)**,i,NOEXPR),
	  IF(RETPARAM(condition$_{s1,\_,sj}$**)**,r,NORET),
	  IF(LOCALVAR(condition$_{s1,\_,sj}$**)**,l,NOVAR)
	  )
	);

method(**E**$_1$,···,**E**$_n$)(**P**$_1$,···,**P**$_m$)
(non tail-recursive)
	MyNode->Statement();
	MyNode->Call(this,(ProcessMethod)&*processClass_*::*method_*,
	  IF(n > 0,new ExprArray(**n**,
	  &P2CPP(E$_1$**)**,
	  ···,
	  &P2CPP(E$_n$**)**
	  ),NULL),
	  IF(m > 0,new ParamArray(**m**,
	  P2CPP(P$_1$**)**,
	  ···,
	  P2CPP(P$_m$**)**
	  ),NULL)
	);

method(**E**$_1$,···,**E**$_n$)()
(tail-recursive)
	MyNode->Statement();
	ret = new Tail((ProcessMethod)&*processClass_*::*method_*,
	  IF(n > 0,new ExprArray(**n**,
	  &P2CPP(E$_1$**)**,
	  ···,
	  &P2CPP(E$_n$**)**
	  ),NULL)
	);

sel
  block$_{s1,\_,sj,1}$
or
.
.
.
or
  block$_{s1,\_,sj,n}$
les
	{
	  ProcessBlock Alternatives[**n**] = {
	  (ProcessBlock) &*processClass_*::*method*_Block_**s**$_1$_…_**s**$_j$_1,
	  …,
	  (ProcessBlock) &*processClass_*::*method*_Block_**s**$_1$_…_**s**$_j$_**n**
	  };
	  ret = MyNode->Select(this,
	  IF(INPUTPARAM(block$_{s1,\_,sj,1}$**)** ∨ ⋯ ∨
	  INPUTPARAM(block$_{s1,\_,sj,n}$**)**,I,NOEXPR),
	  IF(RETPARAM(block$_{s1,\_,sj,1}$**)** ∨ ⋯ ∨
	  RETPARAM(block$_{s1,\_,sj,n}$**)**,r,NORET),
	  IF(LOCALVAR(block$_{s1,\_,sj,1}$**)** ∨ ⋯ ∨
	  LOCALVAR(block$_{s1,\_,sj,n}$**)**,l,NOVAR),
	  n, Alternatives
	  );
	}

[**condition**$_{s1,\_,sj}$] **block**$_{s1,\_,sj}$	Ret = MyNode->Guard(this,
	  IF(INPUTPARAM(condition$_{s1,\_,sj}$**)** ∨
	  INPUTPARAM(block$_{s1,\_,sj}$**)**,i,NOEXPR),
	  IF(RETPARAM(condition$_{s1,\_,sj}$**)** ∨
	  RETPARAM(block$_{s1,\_,sj}$**)**,r,NORET),
	  IF(LOCALVAR(condition$_{s1,\_,sj}$**)** ∨
	  LOCALVAR(block$_{s1,\_,sj}$**)**,l,NOVAR),
	  (CondExpr)&*processClass_*::*method*_Cond_**s**$_1$_…_**s**$_j$,
	  (ProcessBlock)&*processClass_*::*method*_Block_**s**$_1$_…_**s**$_j$
	);

if **E** then
  block$_{s1,\_,sj,1}$
else
  block$_{s1,\_,sj,2}$
fi
	MyNode->Statement();
	if (**P2CPP(E)**.IsTrue())
	{
	  P2CPP(block$_{s1,\_,sj,1}$**)**
	}
	else
	{
	  P2CPP(block$_{s1,\_,sj,2}$**)**
	}

if **E** then
  block$_{s1,\_,sj}$
fi
	MyNode->Statement();
	if (**P2CPP(E)**.IsTrue())
	{
	  P2CPP(block$_{s1,\_,sj}$**)**
	}

while **E** do **block**$_{s1,\_,sj}$ od	MyNode->Statement();
	while (**P2CPP(E)**.IsTrue())
	{
	  P2CPP(block$_{s1,\_,sj}$**)**
	  MyNode->Statement();
	}

block$_{s1,\_,sj,1}$ abort **block**$_{s1,\_,sj,2}$	Ret = MyNode->Abort(this,
	  IF(INPUTPARAM(block$_{s1,\_,sj,1}$**)** ∨
	  INPUTPARAM(block$_{s1,\_,sj,2}$**)**,i,NOEXPR),
	  IF(RETPARAM(block$_{s1,\_,sj,1}$**)** ∨
	  RETPARAM(block$_{s1,\_,sj,2}$**)**,r,NORET),
	  IF(LOCALVAR(block$_{s1,\_,sj,1}$**)** ∨
	  LOCALVAR(block$_{s1,\_,sj,2}$**)**,l,NOVAR),
	  (ProcessBlock)&*processClass_*::*method*_Block_**s**$_1$_…_**s**$_j$_1,
	  (ProcessBlock)&*processClass_*::*method*_Block_**s**$_1$_…_**s**$_j$_2
	);

block$_{s1,\_,sj,1}$ interrupt **block**$_{s1,\_,sj,2}$	Ret = MyNode->Interrupt(this, **IF(INPUTPARAM(block**$_{s1,\_,sj,1}$**)** ∨ **INPUTPARAM(block**$_{s1,\_,sj,2}$**)**,i,NOEXPR), **IF(RETPARAM(block**$_{s1,\_,sj,1}$**)** ∨ **RETPARAM(block**$_{s1,\_,sj,2}$**)**,r,NORET), **IF(LOCALVAR(block**$_{s1,\_,sj,1}$**)** ∨ **LOCALVAR(block**$_{s1,\_,sj,2}$**)**,l,NOVAR), (ProcessBlock)&*processClass_*::*method*_Block_**s**$_1$_…_**s**$_j$_1, (ProcessBlock)&*processClass_*::*method*_Block_**s**$_1$_…_**s**$_j$_2);
delay **E**	MyNode->Delay(**P2CPP(E)**);
skip	MyNode->Statement();

P	**P2CPP(P)**
instVar	self("*instVar*")
inputParam	i("*inputParam*")
retParam	r("*retParam*")
localVar	l("*localVar*")
instParam	self("*instParam*")