3 The Executional Part of an Actor

Introduction

We have seen so far how to define, create, start, and terminate actors in the three languages. Although these operations are important, they do not directly contribute to concurrent matters—they are, after all, merely preliminary actions. Actors are meant to be active and the question therefore should read: *What constitutes the executional part of an actor? How can activity be employed within an actor?* The executional part of an actor shall be the part of an actor that is executed automatically by the actor after starting the actor.

We will, in this chapter, shed light on this issue.

3.1 Procedural Structures

3.1.1 Ada Task Bodies

When a task has finished its activation stage successfully, the execution of the handled sequence of statements of its body begins. As this sequence is not further specified, any legal Ada statement can be used: `loop`s, `case` statements, `if`s, ordinary sequences, subprogram calls. Entry calls and accept statements are also allowed; these will be dealt with in the next chapter. Thus, the executional part of an Ada task consists of the sequence of statements (excluding accept statements) in the task body.

Furthermore, new scope blocks can be introduced so that new data items can be declared. There is no restriction imposed on the variety of such declarations; it is even possible to declare a task inside a task body—actors can be nested in Ada (a task body may also have a declarative part in which these declarations can be executed as well). The important point to keep in mind is sequentiality. Any task body contains a sequence of statements which is executed in a strictly sequential manner. There is exactly one thread of control per task—hence, a task can execute at most one statement at a time.

3.1.2 CHILL Process and Task Bodies

A process instance, once started, begins executing the statements that comprise its body, which is regarded as the executional part of the process instance. A process body contains a data statement list and an action statement list. The former may include declarations and definitions. *There is a static semantic property that forbids a process to be nested within another, however!*

An action statement can be any CHILL action. A started process instance is deemed to be the sequential execution of its contents.

A CHILL task object is somewhat different. Though it possesses its own thread of control after creation and start, it does not execute "voluntarily". Remember that a task body can contain definition statements and the implementations of the procedure specifications introduced in the task specification. There is no room for "code to be run when the task is activated". Instead, a running task object needs a trigger to do useful work—an external actor has to call one of the task's procedures explicitly. At runtime, a location of a task mode simply waits until one of its procedures is called, it then executes this procedure and enters the waiting state again. In the framework of a client/server architecture, we would thus consider a location of task mode to be a *pure server*—i.e., a server that works only on demand. Since a task has no code that is executed automatically, a task cannot naively be used as a substitute for a process. In order to illustrate this matter, we will give an example: Let S be a sequence of statements that is to be performed (1) by a process and (2) by a task. For (1), we simply write:

```
P : PROCESS;  /* to define */
   S;
end P;
⋮
start P;  /* to create and start P, thereby executing S */
```

Compare this to (2), which requires us to define:

```
SYNMODE T = TASK SPEC
   GRANT Do_It;
   Do_It : PROC() END Do_It;
END T;

SYNMODE T = TASK BODY
   Do_It : PROC()
     S;
   END Do_It;
END T;
```

create and start:

```
DCL Executer T;
```

as before with (1). So far, however, nothing important has happened. `Executer` is active but far from doing anything except waiting. To achieve our goal, we must issue the following call:

```
Executer.Do_It();
```

We thus see that the executional part of a CHILL task does not exist since any execution of a task's procedures must be explicitly requested from "outside" the task. This kind of triggering is connected with actor communication, which will be dealt with in the next chapter.

CHILL tasks can, as they are allowed to contain common module components, be nested. Task mode locations have associated with them a single thread of control. As a consequence, although multiple clients may have issued calls to a task's *simple guarded procedures*, at most one—at any time—of these simple guarded procedures is being executed by the task. That is, tasks provide for mutually exclusive access to their components. This holds for regions (see Chapter 5) as well and indeed, a task resembles a region in many ways. A task specification component is defined to be a region specification component and a task body component is a region body component. What, then, is the difference between the two? A task mode location, at runtime, possesses a thread of control whereas a region is a passive entity. That is, if we call a task's component (simple guarded procedure), then the task object itself executes the code of that component. For a region object, a call issued to one of its guarded procedures results in a transfer of control taking place. The caller itself executes the procedure.

Since a task object executes the requested procedure, the caller is allowed to carry on immediately after the call has been issued. When the task object begins executing, the caller can therefore execute the statements following the call—calls to task components are said to be *asynchronous*. A call to a region component, on the other hand, is of *synchronous* nature, meaning that the caller cannot execute the next statement before the call has been finished (this seems logical since the caller executes the code of the component).

A more detailed introduction to regions will be given in Chapter 5. Following below is the detailed semantics of both a call to a task component and a region component (Z200,

1996, [6.7]). If the called procedure is applied to a task mode location, `TL`, the caller performs the following steps:

1. evaluation of the actual parameters
2. send procedure identification, actual parameters, and priority to `TL`
3. continue with the next action

`TL` performs the following steps:

1. receive procedure identification and actual parameters according to priority
2. check of the precondition
3. check of the complete invariant
4. execution of the body of the procedure
5. check of the complete invariant
6. check of the complete postcondition

If the called procedure is applied to a region mode location, `RL`, the caller performs the following steps:

1. evaluation of the actual parameters
2. wait until `RL` is free and lock `RL`
3. check of the precondition
4. check of the complete invariant
5. execution of the body of the procedure
6. check of the complete invariant
7. check of the complete postcondition
8. release `RL`
9. return to the calling point

3.1.3 Bodies of Java Threads

A thread that has been started calls its `run()` method and executes the sequence of statements that is included within that method. The executional part of a Java thread is just that `run()` method. There is nothing special about this sequence—therefore, any kind of Java statement can be used. It is, however, important to realize that a started thread can only execute its `run()` method. Other public methods that the thread offers in its interface cannot be executed by the thread unless, of course, their calls appear in `run()`. We will return to this issue in the next chapter.

Java classes can be nested—hence, an actor can be declared inside another[1]. A Java thread executes the body of its `run()` method sequentially.

3.2 Sequential vs. Concurrent Execution within an Actor

We have mentioned that, in all three languages, the executional part of a running actor has associated with it exactly one thread of control. As a result, whatever statements are contained in the executional parts, they are executed in a strictly sequential manner. Since

[1] these information cannot, at the time of writing this paper, be found in the primary reference for the Java language, (JLS, 1996), but Sun Microsystems has published the so-called Annex D of the manual in the WWW; see `java.sun.com/docs/books/jls/html/1.1Update.html` (July 6, 1998) for more information on changes for Java 1.1

an actor has exactly one executional part, it follows that an actor has associated with it exactly one thread of control.

A Java thread is somewhat special, however. Let `A` and `B` be two Java threads. Assume that `A` offers a method, `S()`, and that both threads are currently executing their `run()` methods (we have a multi-processor machine). So far, there is exactly one thread of control per actor (executing the respective `run()` method). Now imagine that `B`'s `run()` contains the statement `A.S()`. When this call is issued, it is `B` (**not** `A`) that executes `A.S()`. To be even more precise, the call of `A.S()` in `B.run()` is simply an ordinary procedure call. Meanwhile, `A` is still executing its `run()`, but now two threads of control are active within the (single) `Thread` object that represents `A`—`A`'s (executing `A.run()` and `B`'s (executing `A.S()`). Look at Figure 3.1, which depicts this scenario.

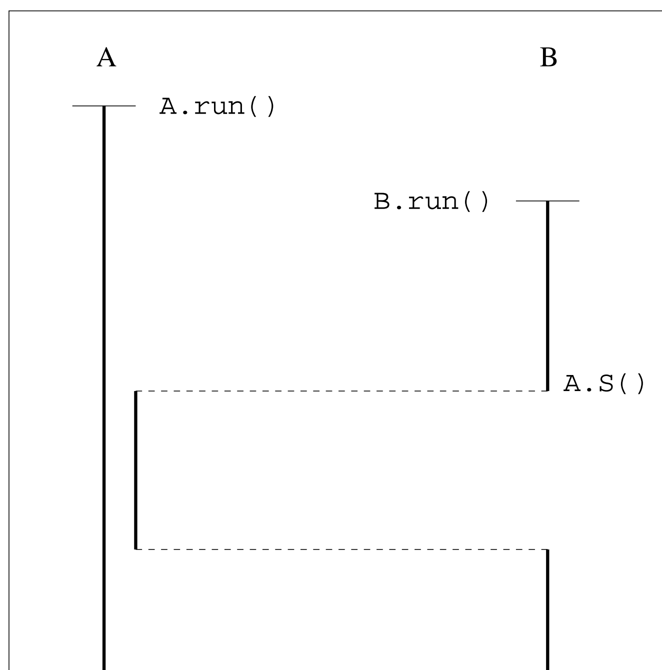

Figure 3.1: Two threads of control are active within one Java actor

Of course, this second thread of control is not active within the executional part of `A`—`A.run()`—but within the thread `A` as a whole. Even if `B.run()` contains the call to `A.run()`, the situation is the same. Calling `A.run()` in `B.run()` still happens to be a simple procedure call. We conclude that there is still exactly one thread of control that "works on behalf" of the executional part of `A`. In fact, there could be an arbitrary number of threads of control executing code that belongs to `A` by having further threads call `A.S()`. This is as much as if a procedure in an Ada package or CHILL module were called by multiple tasks. However, and this is the special point of the discussion, neither an Ada package nor a CHILL module can be considered as an actor (since they are passive). A Java thread combines the feature of being a module with that of being an actor and thus having a thread of control.

In Ada, an interesting issue regarding threads of control arises when we employ *asynchronous transfer of control (ATC)*. Unfortunately, we cannot, in this chapter, address these ideas since ATC remains to be introduced in the next chapter.

Summary and Comparison

We have taken, in this chapter, the opportunity to study the executional parts of actors in our three languages in more detail. We saw that these parts can incorporate easily almost any kind of statement the actor should execute. The languages are very similar with respect

to that issue—almost any statement that can occur elsewhere is allowed to be placed inside an actor. Nesting of actors is possible with Ada tasks, CHILL tasks (but not processes), and Java classes.

This chapter is somewhat special in that it contains material that would not fit elsewhere: We pointed out the difference between a CHILL process and a CHILL task. Following that, we turned to the somewhat subtle topic of concurrency within an actor. We could rank the languages with respect to that matter: Ada, CHILL, Java. This is an ordered list and a language precedes another whenever the number of threads of control an actor can have in this language is less than or equal to that in the other language. All Ada and CHILL actors are single threaded. In Java, the situation is different: an arbitrary number of thread objects (and hence, threads of control) can simultaneously be executing code belonging to another thread object `O`.