Coroutine Sequencing in BETA

Bent Bruun Kristensen
Ole Lehrmann Madsen
Birger Møller-Pedersen
Kristen Nygaard

DAIMI PB – 235
January 1989
2nd Edition
Coroutine Sequencing in BETA *

Bent Bruun Kristensen  
Inst. of Electronic Systems, Aalborg University Centre  
Birger Møller-Pedersen  
Norwegian Computing Center  
Ole Lehrmann Madsen 
Computer Science Department, Aarhus University  
Kristen Nygaard  
Institute of Informatics, Oslo University  

November 16, 1987

Abstract

In the object-oriented perspective a program execution is viewed as a physical model of some real or imaginary part of the world. A programming language supporting the object-oriented perspective must therefore contain comprehensive facilities for modelling phenomena and concepts from the application domain. In object-oriented programming objects model physical material and classes model concepts. By means of subclassing and virtual attributes it is possible to model a hierarchical classification of objects. Many applications in the real world consist of objects carrying out sequential processes. Coroutines may be used for modelling objects that alternate between a number of sequential processes. This paper describes coroutines in BETA.

*Presented at Hawaii International Conference on System Sciences — 21, January 5-8, 1988
1 Introduction

The most common language construct associated with object-oriented programming is subclassing, often called inheritance. Subclassing supports a hierarchical classification of objects by making it possible to isolate common properties in general superclasses. Together with the notion of virtual attributes (similar to methods in Smalltalk [Smalltalk 84]) these mechanisms have been recognized as useful in a large number of applications.

There is however more to object-oriented programming than just inheritance. In the object-oriented perspective a program execution is conceived as a physical model of some real or imaginary part of the world. Many applications in the real world consist of objects that carry out individual action sequences. That is, an application often consists of a collection of objects, each carrying out a sequential process. When creating models of such applications it is therefore desirable that objects with individual action sequences can be described in the programming language in use. Consequently a language that supports the object-oriented perspective should have constructs for describing objects with individual action sequences.

Simula 67 [Simula 68] supports quasi-parallel sequencing by means of coroutines. In Smalltalk it is to a limited extent possible to model this by means of the classes Process, Semaphore and ProcessScheduler.

In BETA [BETA 87b], objects with individual action sequences may be modelled in two ways, either as objects being executed concurrently or as objects being executed in alternation.

Objects are “state machines” in the sense that the result of a remote procedure call (method invocation) may depend on the state of the variables of the object. For objects that are coroutines, the state may include a point of execution. In general such an execution state involves a stack of procedure activations currently called. The possibility of saving the state of execution makes coroutines useful for a large number of applications. These applications may be grouped as follows:

- Coroutines may be used to create an illusion of concurrency. A major example of this is discrete event simulation ([Simula 68]). In Modula-2 ([Wirth 82]) coroutines are directly used to simulate concurrent processes. The basic scheduling of coroutines is usually
explicit, since a coroutine relinquishing control names the coroutine to take over. In Simula and Modula-2 it is possible to eliminate the explicit scheduling by construction of a coroutine scheduler. By using coroutines, mutual exclusion is always guaranteed\(^1\). The order of scheduling may however by difficult to predict.

- With respect to modelling of real life phenomena the main motivation for coroutines is to model objects that perform alternating activities. The alternation between activities may be deterministic in the sense that sequencing is decided by the object itself. The shifts between activities may be triggered by events performed by other concurrent objects leading to nondeterministic alternation.

- Coroutines are useful whenever an algorithm is best understood and described as a set of interlocked partial algorithms ([Wang & Dahl 71]), including backtracking and pattern matching.

- A generator is a coroutine that is capable of producing a sequence of values. A value is produced for each invocation of the coroutine. Such a coroutine is characterized by always returning to its caller. Icon ([Griswold et al. 81]) is an example of a language that supports generators.

In [Marlin 80] a distinction is made between two types of coroutine sequencing. The first kind of coroutine, the implicit sequencing kind, communicates only via first-in-first-out queues and there is no explicit transfer of control between the coroutines. Call-by-need parameters, lazy evaluation, streams (as in [Scheme]) and the system described in [Kahn & MacQueen 77] are examples of this kind of coroutines.

For the second kind of coroutine, the explicit sequencing kind, it is possible to transfer control explicitly from one coroutine to another.

Only few programming languages support explicit coroutine sequencing. Simula is one of the few languages that offers an advanced design. In Simula there is a distinction between semi-coroutines and symmetric coroutines. A semi-coroutine is executed by means of the new- or call-imperative; a subsequent detach returns execution to the caller. Symmetric coroutines are always explicitly scheduled by means of the resume-imperative.

\(^1\)This is not the case with Modula-2 coroutines used as interrupt handlers.
Unfortunately, the details of coroutine sequencing in Simula are very complicated. The problem is to understand how semi-coroutines, symmetric coroutines and prefixed blocks are integrated. This means that even experienced Simula programmers may have difficulties in figuring out what is going on in a program using coroutines. The details of Simula’s coroutines sequencing is described in [Simula 68].

A simplified version of Simula’s coroutine mechanism has been presented in [Dahl & Hoare 72]. A formal description of part of the coroutine mechanism has been presented in [Wang & Dahl 71]. This formalization has been further elaborated in [Lindstrom & Soffa 81]. In [Wang 82] it was shown that the semantics of Simula’s coroutine mechanism was inconsistent. The problem was that deallocation of block instances could not be performed as stated by the original language definition. It is argued that the simple model of [Wang & Dahl 71] cannot cope with full Simula. Wang presents a detailed analysis of Simula’s coroutine mechanism and gives certain proposals for changes to Simula. These proposals have since then lead to a change in the semantics of Simula ([Simula 87]).

Explicit coroutine sequencing in the form of symmetric coroutines is also present in Modula-2. According to [Henry 87] there are severe problems with the definition of the coroutine mechanism in Modula-2.

For a further discussion of the history and motivation for coroutines see [Marlin 80] and [Horowitz 83].

The purpose of this paper is to present the mechanisms for coroutine sequencing in BETA. Coroutines in BETA are similar to semi-coroutines in Simula. The BETA constructs are simpler and more general than those of Simula. In addition BETA offers the possibility of including parameters when calling coroutines. The BETA constructs for semi-coroutines may be used to define a set of attributes that model Simula’s symmetric coroutines. For people with a background in Smalltalk, and Lisp extensions such as Flavors and Loops this paper may also be read as an introduction to the basic principles behind coroutine sequencing in Simula and BETA.

Since this paper only treats coroutine sequencing in BETA there are major parts of BETA which will not be described. Certain parts will be used without a detailed explanation. For a more detailed account, the reader is referred to [BETA 87b].
2 The Wang and Dahl Model

The approach suggested by Wang and Dahl for characterizing coroutines will be used as a basis. The formulation below is highly influenced by [Lindstrom & Soffa 81]. A program execution consists at a given moment of a set of objects\(^2\), \(S\), augmented by a special element, \(P^*\), representing the executing environment (the processor). Each object may execute a sequence of actions. Objects may execute other objects giving rise to a relation called the dynamic link. The dynamic link relation may change during the program execution. In order to represent this relation, each object contains a hidden variable named SC (sequence control). These notions may be summarized as follows:

S the set of objects in existence at any moment.

\(x\).SC, for \(x \in S\) the return link of \(x\), a pair of the form
\[\text{[ip: return code pointer, ep: calling object]}\]

\(D\) a function \(S \to S\) denoting dynamic enclosure with \(D(x) \equiv x\).SC.ep.\(

\(P^*\) the processor, in \(S\) by extension. By special convention, \(P^*.SC.ep \equiv D(P^*) \equiv \text{the currently operating object, and } P^*.SC.ip \equiv \text{the program counter of } P^*\).

\(\to\) a binary relation on \(S\), defined to be \(x \to y \equiv x \neq P^* \land D(x) = y\); the transitive closure is denoted \(\to^+\); the transitive and reflexive closure is denoted \(\to^*\).

\(OC\) the set of instances dynamically linked from \(D(P^*)\), that is, \(\{x | D(P^*) \to^* y\}\), is termed the operating chain (OC). An object \(x\) is said to be active iff \(x \in OC\).

It is now possible to define a class of possible control events: This includes creation of objects, invocation of objects, termination of objects, and the control exchange actions swap\((x)\) and rotate\((x, y)\).

1. Invariant: Initially \(D(P^*) = P^*\). At any time during execution there are one or more lists of objects linked by the dynamic link. An object is a member of exactly one such list. The list linked

\(^2\)Wang and Dahl consider the set of dynamic procedure instances (activations). In BETA there is no distinction between instances of a procedure and instances of a class (objects).
from $P^*$ describes the active objects. $P^*.SC$ identifies the currently executing object. The remaining lists describe currently suspended objects. The state of execution is described in Figure 1: $y$ is the currently executing object and $z$ is the outermost object.

2. **Creation:** Let $C$ be a class as in Simula. The result of creation of an instance of $C$ is an object $x$, where $x.SC = [x.firstAction, x]$. $x.SC$ saves the first action of $x$ to be executed. See Figure 2.

3. **swap($x$):** This event causes the SC variables of $x$ and $P^*$ to have their values interchanged.

   $x.SC$ $P^*.SC$

If $x \in OC$, the effect of $\text{swap}(x)$ is a return of control to its caller without termination of $x$ (i.e. $x$ temporarily suspends execution). $x.SC$ saves the reactivation point of $x$.

If $x \notin OC$, the effect of $\text{swap}(x)$ is the establishment of $D(P^*)$ as the current caller of $x$ and the resumption of $x$ at its reactivation point (i.e., $x$ is "called"). Figure 3 illustrates the effect of $\text{swap}(x)$. The transition from state (a) to state (b) or vice versa is accomplished by the same action.
4. \textbf{rotate}(x, y): This event causes the SC variables of \( x, y, \) and \( P^* \) to have their values permuted as indicated below:

\[
\begin{align*}
\text{P}^\ast \cdot \text{SC} & \\
\times \cdot \text{SC} & \\
y \cdot \text{SC} & \\
\end{align*}
\]

The effect of \textbf{rotate}(x, y) is equivalent to \textbf{swap}(x) followed immediately by \textbf{swap}(y) in an indivisible action. See Figure 4.

5. \textit{Termination:} Assume \( D(P^*) = x \neq P^* \). The termination of \( x \) causes the event \textbf{swap}(x) to appear. In addition \( x \cdot SC \cdot ip = \text{abort} \). A subsequent \textbf{swap}(x) will then result in an \textbf{abort} event, which terminates the program execution.

3 Components

The BETA construct corresponding to a class in Simula and Smalltalk is called a \textit{pattern}. A declaration of a pattern has the form:
$P: P0(\# Dcl_1; Dcl_2; \ldots Dcl_n,$

$\quad$ enter $In$

$\quad$ do $Imp$

$\quad$ exit $Out$

$\#)$

where $P0$ is the super-pattern of $P$. $Dcl_1$, $Dcl_2$, $\ldots$, $Dcl_n$ are declarations of attributes. An attribute may be either a reference to an object or a pattern. $In$ corresponds to input parameters. $Imp$ (called the $do$-part) is an imperative describing actions to be executed when the object is executed as a procedure, coroutine or concurrent system. $Out$ corresponds to output parameters.

The pattern is an abstraction mechanism, which is a unification of classes, procedures, functions and types. Instance of a pattern, called objects, may be used as variables, data structures, procedure/function activations, coroutines and concurrent systems. BETA has three kinds of objects: system, component and item. The kind of an object specifies how the object can be used. Objects of kind component may be used as coroutines. In the following it is described how to create and execute components. The term object will be used whenever we describe something that is true for all three kinds of objects. When a kind, like component, is explicitly mentioned, the explanation is only valid for that kind of objects.

**Creation:** The declaration

$$R: @(P);$$

describes that a component instance of $P$ is created, giving rise to a creation event. $R$ is a so-called static reference that will constantly denote the newly created $P$-component.

**Attachment:** An imperative like:

$$R$$

where $R \not\in OC$ implies that the event swap($R$) takes place. The component executing $R$ is said to attach $R$.

**Suspension:** Assume that $R \in OC$ and that $R$ is the currently operating component ($D(P^*) = R$). The imperative

\begin{verbatim}
suspend
\end{verbatim}
(executed by $R$) implies that the event $\text{swap}(R)$ takes place. $R$ is now said to be suspended.

Termination: If the currently operating component finishes execution of the imperative in its do-part, termination of the component will take place. This implies execution of an implicit suspend. A subsequent attachment will result in an $\text{abort}$ event.

A $\text{swap}$ event gives rise to either coroutine attachment or suspension. As it appears these two cases are denoted differently in BETA\textsuperscript{3}.

Attachment of $R$ implies that the component denoted by $R$ will be executed. This means that the actions described by the imperatives in the do-part of $R$ are executed. The execution of the component continues until the component executes a suspend imperative. This will return the control to the point of the attachment. A subsequent execution (attachment) of the component will resume the component after the suspend imperative. This pattern may be continued until the component has completed execution of its do-part.

Consider the example in Figure 5. $\text{TrafficLight}$ describes components that when executed alternate between two states $\text{red}$, $\text{green}$. The $\text{Controller}$ component initializes the state of $\text{North}$ to $\text{red}$ and the state of $\text{South}$ to $\text{green}$. It then repeatedly waits for some time, and then switches the lights.

In the following a few more BETA constructs are explained in order to understand the details of the example.

- The component $\text{Controller}$ is described directly without referring to a pattern. An object being described directly is called a singular object.

- The main program itself

  $$(\# \ldots \text{doController}\#)$$

  describes a singular component which is the outermost component being attached by the processor $P^*$.\footnote{This is also the case in Simula where attachment is denoted $\text{call}(R)$ and suspension is denoted $\text{detach}$. In Simula, creation of a coroutine is followed by an immediate attachment.}

- The “variable” $\text{state}$ is a static reference denoting an instance of the pattern $\text{Color}$. The $\text{Color}$ instance is an object of kind $\text{item}$ whereas
Figure 5: Example of components

all the other objects mentioned so far are components. An item is not a coroutine. In this example the Color instance is used as an ordinary variable. Below the role of items will be further explained.

- The construct

\[ \text{Cycle}(\# \text{do Imp'}\#) \]

implies that the imperative Imp' \(^4\) is executed forever. Cycle is an example of a pattern used for defining a control structure. For a further description of combining action parts in BETA, see [BETA 87a].

\(^4\)The construct \(\text{Cycle}(\# \text{do Imp'}\#)\) is similar to a prefixed block in Simula. In Simula prefixed blocks play a major role in quasi-parallel sequencing. This is not the case in BETA.
Components with parameters

Components may have enter/exit parameters. Prior to the attachment of a component, a value may be assigned to the enter part of the component. When a component suspends execution or terminates, a value may be assigned from its exit part. If \( R \) is a component having enter/exit parts, then attachment of \( R \) with parameter transfer has the form:

\[
X \rightarrow R \rightarrow Y
\]

where \( X \) may be an expression or a list of expressions and \( Y \) may be a "variable" or a list of variables. The value of \( X \) is assigned to the enter-part of \( R \). Then the component \( R \) is attached — that is, execution of \( R \) is resumed. Finally, when \( R \) suspends execution the exit part of \( R \) is assigned to \( Y \).

In Figure 6 an example of a component having enter/exit parts is given. The component \texttt{Factorial} computes \( N! \). A call of the form \( E \rightarrow \text{Factorial} \rightarrow F \) returns \( E! \) in \( F \). A subsequent call \( \text{Factorial} \rightarrow F \) returns \( (E + 1)! \). At any time a new enter parameter may be given. \texttt{Factorial} values computed previously are saved in a table. That is, each factorial value is only computed once. \texttt{Factorial} is an example of a generator that computes a sequence of values.

4 Components and Items

The examples so far have showed coroutines that do not execute procedures as part of their actions. Such coroutines may be simulated using simple variables, since there is only a finite set of suspension points. If coroutines are combined with (recursive) procedure calls it is much more complicated to simulate the state of execution at suspension points. In this section the BETA constructs for combining coroutines and procedures are described.

A pattern may be used to create objects that behave like procedure activations. Such objects are of kind \textit{item}. Below the semantics of creation and execution of items is described.

Creation and attachment: The imperative

\[
X \rightarrow \&P \rightarrow Y
\]
\textit{Factorial: \texttt{factorial}} \{a singular component\}

\texttt{# T: \{100\}@\texttt{Integer};}
\{an array \texttt{T[1]}, \ldots, \texttt{T[100]} of integer items\}
\texttt{N, Top: \texttt{@Integer};}
\texttt{enter N}
\texttt{do 1 \rightarrow Top \rightarrow T[1];}
\texttt{Cycle}
\texttt{  (# do}

\texttt{(if (Top < N) \texttt{// True then}}
\{Compute and save \texttt{(Top + 1)! \ldots N!}\}
\texttt{(for i \texttt{in}[Top + 1, N] \texttt{repeat}}
\{\texttt{T[i - 1] = (i - 1)!}\}
\texttt{T[i - 1] \ast i \rightarrow T[i]}
\{\texttt{T[i] = i!}\}
\texttt{for);}
\texttt{N \rightarrow Top}
\texttt{if);}\;
\texttt{N + 1 \rightarrow N;}
\texttt{suspend} \{\texttt{suspend and exit T[N - 1]}\}
\{\texttt{A new value may have been assigned}\}
\{\texttt{to N through enter }\}
\texttt{#} \}
\texttt{exit T[N - 1]}
\texttt{#);}

\texttt{F: \texttt{@Integer}}
\texttt{do 4 \rightarrow \texttt{factorial} \rightarrow F; \{F = 4!\}}
\{This execution of \texttt{factorial} will result in\}
\{computation of 1!, 2!, 3! and 4!\}

\texttt{Factorial \rightarrow F; \{F = 5!\}}
\{Here 5! was computed\}

\texttt{3 \rightarrow \texttt{factorial} \rightarrow F; \{F = 3!\}}
\{No new factorials were computed by this call\}
\texttt{#}

\textbf{Figure 6: A generator for factorials}

describes creation and execution of a \textit{P}-item. A \textit{P} item, say \textit{E}, is
created by a creation event as described in section 2. The value
of \textit{X} is then assigned to the enter part of \textit{E}; a \texttt{swap(\textit{E})} event
is executed; finally when \textit{E} is terminated, the exit part of \textit{E} is
assigned to $Y$.

**Termination:** Termination of an item takes place as for a component.

**Suspension:** An item may not be suspended like a component. In section 2 the notion of *currently operating object* is defined. In section 3 the semantics of suspend is defined relative to the *currently operating component*. These definitions shall be made precise here:

- The *currently operating object* is the object denoted by $P^*SC$. The currently operating object may be either an item or a component.
- The *currently operating component* is a component $C$ satisfying

\[ x_n \rightarrow^* C \land P^*SC.ep = x_n \land \text{if } x_n \rightarrow^+ x_i \rightarrow^* C \text{ then } x_i \text{ is an item} \]

I.e. the currently operating component is the "leftmost" component in $OC$.

Consider Figure 7. Assume that $C$ and $C1$ are components and that $x_1, \ldots, x_n$ and $y$ are all items. $C$ is the currently operating component and $x_n$ is the currently operating object. According to the definition of suspend in section 3, an execution of suspend by $x_n$ will then imply a $\text{swap}(C)$ resulting in a transition from state (a) to state (b).

The execution states described in Figure 7 may be generated by the BETA program in Figure 8.
\(X: (\#
  \text{do } \{\text{if } i = n \text{ then execution state}\}
  \{\text{is described by Figure 7(a)}\}
  \text{suspend;}
  \&X;
  \{\text{create and execute an item } x_i\}
  \{\text{this is a recursive "call" of } X\}
\)#);

\(C: @ | (\# \text{ do } \ldots; \&X \ldots#);\)

\(C1: @ | \)
\(# Y:\)
\(# \text{ do } (\text{for } i : \text{in}[1, n] \text{ repeat } C \text{ for})\)
\(# \{\text{execution state corresponds to Figure 7(b)}\}\)
\(#\)
\(\text{do } \ldots \&Y; \ldots#\)
\(#\)
\(\text{do } \ldots C1 \ldots\)
\(#\)

Figure 8: A program corresponding to Figure 7

The example in Figure 9 is a classical example of using coroutines. The program describes a merge of two binary search trees. The attribute \textit{Traverse} performs an inorder traversal of the tree. \textit{Traverse} is a component that will suspend and exit the elements in the nodes visited during the traversal. The main program starts by executing \textit{Traverse} for each of the trees \textit{b1} and \textit{b2}. The smallest element of \textit{b1} will then be delivered in \textit{e1} and the smallest element of \textit{b2} will be delivered in \textit{e2}. The \textit{merge} loop will then print the smallest of the two elements. If e.g. \textit{e1} is the smallest then \textit{e1} is printed and \textit{b1}. \textit{Traverse} will exit the next element of \textit{b1}. This continues until there are no more elements in the two trees.

A few more parts of BETA need to be explained:

- A declaration of the form

  \(b1: @\textit{BinTree}\)

  describes creation of a \textit{BinTree}-item. \textit{b1} is a static reference denoting this newly created item. Notice the similarity with creation of
static components.

- A declaration of the form
  \[ \text{root: } \uparrow \text{node} \]
  describes a \textit{dynamic reference} to an item. A dynamic reference may
denote different objects during the program execution. Initially a
dynamic reference has the value \text{NONE}, i.e. no object is denoted.
A dynamic reference is similar to a qualified reference in Simula
and an instance variable in Smalltalk.

- A dynamic reference may be given a value in the following way: \(^5\)
  \[ \&\text{node} \rightarrow \text{root} \]
  The expression \&\text{node} \rightarrow \text{root} describes creation of an instance of \textit{node}
and the value of the expression is a reference to this instance.

- Objects are in general denoted by references and in general ex-
pressions of the form \( R \) describe the object denoted by \( R \). If the
reference itself is denoted, a box (\( \Box \)) is appended to the expression
as in \( R \Box \).

5 Abstract Super-Patterns

A major design goal for BETA has been to design a language with a small
number of basic, but general primitives. In addition much emphasis has
been put into design of powerful abstraction mechanisms. In this way it is
possible to define more specialized constructs. Object-oriented languages
provide powerful constructs for defining abstract super-patterns\(^6\) that
describe the general properties of a class of (partial) program executions.

Class \textit{Simulation} of Simula is a classical example of an abstract super-
class. It introduces the notions of processes and event notices along with
a scheduling mechanism. Simulation programs may then be expressed as
specializations of class Simulation.

In this section examples of defining abstract super-patterns in BETA
will be given. This will include modelling of symmetric coroutines in the

\(^5\)Corresponding to \textit{root} \leftarrow \textit{node New} in Smalltalk.

\(^6\)In Smalltalk terminology, an abstract super-pattern (super-class) is a pattern (class) that is
only used as a super-pattern. That is, no instances are created.
style of Simula and illusion of concurrent programming. First a short
description of sub-patterns and virtual patterns will be given.

Consider the two patterns in Figure 10. \( PP \) is a sub-pattern of \( P \).
This implies that instances of \( PP \) have attributes \( a, b, \) and \( m1 \). The
actions taking place when executing an instance of \( PP \) is a combination
of the do-part of \( P \) and the do-part of \( PP \). Execution starts with the
do-part of \( P \). An occurrence of inner implies execution of the do-part of
\( PP \). Execution of an instance of \( PP \) will then result in execution of \( A1, A2, \) and \( A3 \).

The attribute \( m1 \) of \( P \) is a virtual pattern. The description of a
virtual pattern may be extended in sub-patterns of \( P \). The pattern \( PP \)
extends the description of \( m1 \). \( m1 \) is extended to be a sub-pattern of
the \( m1 \) pattern described in \( P \). Let \( X \) be an instance of \( PP \). Execution
of \( X.m1 \) will then imply execution of \( I1, I3, \) and \( I2 \). Both \( I3 \) and \( A3 \)
may contain an inner allowing further specialization of \( PP \) and \( m1 \). See
[BETA 87b,BETA 87a] for a more detailed description of sub-patterns
and virtual patterns.

5.1 Symmetric Coroutines

The sequencing of components as described in the previous sections cor-
responds to semi-coroutines of Simula. In this section it will be shown
how to model Simula's symmetric coroutines.

The pattern SymmetricCoroutineSystem of Figure 11 is an abstract
super-pattern that describes the general properties of a symmetric coro-
This is done by means of the
resume attribute. Resume implements the rotate primitive. Note that
Resume is a virtual pattern. This means that it is possible to extend the
definition of Resume in sub-patterns of SymmetricCoroutine.

A SymmetricCoroutineSystem terminates when the active Symmetric-
Coroutine terminates execution without using resume. This may happen
either by executing a suspend or by terminating its action part.

In Figure 12, an example of a program using the pattern SymmetricCoroutineSystem is given. The problem to be solved ([Grune 77]) is to copy characters from input to output. Any occurrence of a string 'aa' must be converted to 'b'. Similarly a string 'bb' must be converted to 'c'. The latter includes 'a's converted to 'b's. A string 'abcaadbbbeaabb' will thus be converted into 'abcdbcec'. The Converter terminates by means of suspend when a newline character (\n) is recognized at the outermost level of DoubleBtoC. Notice that the description of the Resume attribute has been extended to include an enter parameter in DoubleBtoC.

5.2 Quasi-parallel Systems

In this section it is shown how to simulate concurrency by means of coroutines. The example is inspired by the Process module in [Wirth 82]. In Figure 13 an abstract super-pattern for defining quasi-parallel sequencing is presented. A QuasiParallelSystem defines an abstract super-pattern, Process, for defining coroutines that may take part in the quasi-parallel sequencing. A coroutine that is to take part in the scheduling must be a specialization (sub-pattern) of the pattern Process. Instances of sub-patterns of Process are hereafter called processes.

The pattern ProcessQueue defines a queue of processes. All active processes are placed in an instance of ProcessQueue called Active. Each time a process suspends execution, a new process is selected from this queue.

Communication among processes is synchronized by means of signals (c.f. [Wirth 82]). A process may send a signal and it may wait for (some other process sending) a signal. In the example a signal is implemented as a ProcessQueue.

In Figure 14, the classical producer/consumer system is implemented as a quasi-parallel system. Patterns describing the behavior of producers and consumers are defined. Producers and consumers communicate by means of the buffer B and the signals notFull and notEmpty. A producer component P1 and a consumer component C1 are declared.
6 Conclusion

More than 15 years of experience with Simula has demonstrated that coroutines are an important mechanism in their own right. And since Simula is an object-oriented language, coroutines are certainly useful within an object-oriented framework. A major difference between Simula and Smalltalk is that Smalltalk classes do not have a do-part. It should however be straightforward to reinvent the do-part of Smalltalk classes and thereby allowing Smalltalk objects to be active coroutines.

The experience with Simulas coroutine mechanism has been the starting point for the BETA design. As mentioned in the introduction the details of Simulas coroutine mechanism are very hard to understand and inconsistencies in the semantics have been detected recently. However, in most Simula programs these problems do not show up. Another problem with Simulas coroutine mechanism was the inability to transfer parameters when calling a coroutine. The lack of parameters makes it clumsy to implement generators in Simula, since parameters must be transferred by means of global variables.

In the design of BETA, it has been attempted to include a simple and general coroutine mechanism that keeps the advantages of Simula. The simple mechanism together with a powerful abstraction mechanism makes it possible to implement a wide variety of sequencing schemes. The symmetric coroutines and quasi-parallel systems in section 5 are examples of this. BETA adds nothing to the basic principles of coroutine sequencing used in Simula. However, the technical details of coroutine sequencing in BETA are much simpler than those of Simula. In addition, coroutines in BETA may have parameters. This makes it easier to use BETA coroutines as generators. Coroutine (components) calls appear like procedure calls (items) whereby a high degree of uniformity between procedures and coroutines is obtained.

The arrival of Modula-2 has resulted in a renaissance for coroutines. However, coroutines in Modula-2 are considered low-level facilities for implementing concurrent processes. According to [Henry 87] this has implied that the status of coroutines in Modula-2 is unclear. In BETA coroutines are a well integrated part of the language.

A major reason for introducing coroutines in BETA is for modelling objects that alternate between a number of sequential processes (tasks). Alternation should not be confused with true concurrency, where a num-
ber of tasks take place at the same time. In alternation at most one of the tasks takes place at a given time.

Deterministic alternation is the situation where the object decides by itself how to alternate between the different tasks. Nondeterministic alternation is the situation where external events cause the object to shift to another task.

Consider a bureau with a number of agents. Each agent serves a number of costumers and has a file for each costumer. A task for an agent is to process a costumer file. During a work day the agent alternates between the tasks processing the costumer files. The agent will most of the time decide the order of the tasks. However, external events such as telephone calls, may force the agent to shift task.

Coroutine sequencing as treated in this paper supports modelling of deterministic alternation. In [BETA 87b] it is shown how to model nondeterministic alternation in the context of truly concurrent objects. Modula-2 coroutines used as interrupt handlers may be viewed as nondeterministic alternation.

The BETA coroutine mechanism has been presented in terms of the Wang and Dahl model. The model is operational and may be viewed as an abstract implementation. In fact the current implementation in BETA follows this model very closely. It may be argued that a more abstract and less operational model should be used for explaining the semantics. However, from an object-oriented perspective, where coroutines are viewed as models of alternating sequential processes from the real world, the model appears quite natural. The model has not been selected for its mathematical properties.

Acknowledgement. We wish to thank Jørgen Lindskov Knudsen, Boris Magnusson and the anonymous referees for many helpful comments on this paper. This work has been supported by The Danish Natural Science Research Council, FTU grant no. 5.17.5.1.25 and The Royal Norwegian Council for Scientific and Industrial Research, grant no. ED 0223.16641.

References


[BETA 87a] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Classi-


(###
BinTree:
(#
    Node: {The nodes of the binary tree}
    (# elem: @Integer;
        left, right: ↑ Node;
    #);
    root: ↑ Node;

Traverse: @ |
(# next: @Integer;

Scan:
(# current: ↑ Node;
    enter current
    do (if (current = NONE) // false then
        current.left → &Scan;
        current.elem → next;
        suspend;
        current.right → &Scan;
    if) #);
    do root → &Scan;
    maxInt → next; Cycle(# do suspend #);
    {exit maxInt hereafter}
    exit next
    #) {Traverse}
do …
#); {BinTree}

b1, b2: @BinTree;
e1, e2: @Integer
do …
b1.Traverse → e1;
b2.Traverse → e2;

merge: Cycle
(# ….
    do (if (el = MaxInt) ∧ (e2 = MaxInt)
        // True then leave merge if);
    (if (el < e2)
        // true then
        el → print; b1.Traverse → e1
        // False then
        e2 → print; b2.Traverse → e2
        if);
    #);
…

Figure 9: Merge components
\[ P: (\# a: \&\text{Integer}; \]
\[ m1 \ll (# \text{do I1; inner; I2#}) \]
\[ \text{do A1; inner; A2} \]
\[ \#); \]
\[ PP: P(\# b: \&\text{Integer} \]
\[ m1 \ll (# \text{do I3#}) \]
\[ \text{do A3} \]
\[ \#) \]

Figure 10: Sub-pattern and virtual pattern
SymmetricCoroutineSystem:
(#
SymmetricCoroutine: {Abstract super pattern}
(#
    Resume <<
    (# do this SymmetricCoroutine \rightarrow next;\{save "self" in next\}
    {for subsequent attach by ScheduleLoop}
    suspend {suspend caller}
    #)
    do inner
    #);

Run: {start of initial SymmetricCoroutine}
(#
enter next\{global reference declared below\}
do ScheduleLoop:
    Cycle
    (# active: \rightarrow | SymmetricCoroutine \{Currently operating component\}
do (if (next \rightarrow active)
    // NONE then leave ScheduleLoop
    if);
    NONE \rightarrow next;
    active {attach next SymmetricCoroutine} \{terminates when active executes\}
    {resume, suspend or terminates}
    #)

next: \rightarrow | SymmetricCoroutine;
    {next SymmetricCoroutine to be resumed}
do inner
#); {SymmetricCoroutineSystem}

Figure 11: A general symmetric coroutine system
Converter: @ | SymmetricCoroutineSystem

('#
DoubleAtoB: @ | SymmetricCoroutine
(# ch: @Char
  do Cycle
    (# do GetChar \to ch;
      (if ch
        // 'a' then
        GetChar \to ch
        (if ch
          // 'a' then
          'b' \to DoubleBtoC. Resume
        else
          'a' \to DoubleBtoC. Resume;
          ch \to DoubleBtoC. Resume
        if)
      else
        ch \to DoubleBtoC. Resume
      if) #)

DoubleBtoC: @ | SymmetricCoroutine
(# ch: @Char;
  Resume ::= ( #enter ch#)
  do Cycle
    (# do (if ch
      // 'b' then
      DoubleAtoB. Resume;
      (if ch
        // 'b' then
        'c' \to PutChar
      else
        'b' \to PutChar;
        ch \to PutChar
      if)
      // nl then suspend
      else ch \to PutChar
    if);
    DoubleAtoB. Resume
  if) #)
  do DoubleAtoB□ \to Run
  #);

Figure 12: A SymmetricCoroutineSystem
QuasiParallelSystem:
(# ProcessQueue:
  (# Insert: {Insert a process}
   {Insert of NONE has no effect}
   Next: {Exit and remove some process}
   {If the queue is empty, NONE is returned}
   Remove: {Remove a specific process}
#);
Active: @ProcessQueue; {The active processes}

Process: {General quasi-parallel processes}
(#
  Wait: {Make this Process wait for a send to S}
  (# S: ↑ ProcessQueue
    enter S\)
    do this Process\ → S.Insert;
    this Process\ → Active.Remove;
    suspend
#);

Send: {Activate a process from S}
( # S: ↑ ProcessQueue
  enter S\)
  do S.Next → Active.Insert;
  suspend
#);
 do inner
#); {Process}

Run: {The scheduler}
(# Ap: ↑ | Process
  {currently active Process}
 do ScheduleLoop :
  Cycle
  (#
  do (if (Active.Next → Ap\)
    // NONE then leave ScheduleLoop
    if);
    Ap\ → Active.Insert; {Ap is still active}
    Ap {Attach Ap}
  #)#)
 do inner
#)

Figure 13: A general quasi-parallel system

25
ProducerConsumer: © | QuasiParallelSystem
(# B: ®Buffer;
   notFull, notEmpty: ®ProcessQueue; {Signals}

Producer: Process
(# Deposit:
   (# E: ®BufferElement
    enter E
    do (if B.Full // True
         then notFull E © → Wait if);
     E → B.Put;
     notEmpty E © → Send
    #)
doi inner
#);

Consumer: Process
(# Fetch:
   (# E: ®BufferElement
    do (if B.Empty // True
         then notEmpty E © → Wait if);
     B.Get → E;
     notFull E © → Send;
    exit E
    #)
doi inner
#);

P1: ® | Producer(# ...E1 → Deposit;...#);
C1: ® | Consumer(# ...Fetch → E1;...#);
do P1 © → Active.Insert; C1 © → Active.Insert;
&Run
#)

Figure 14: A producer/consumer system