The Design of a new Simula System

by

Øystein Myhre Andersen

Background.

The 50th anniversary of Simula, the first object-oriented programming language, was held at the Department of Informatics, University of Oslo on 27th September 2017. James Gosling, the inventor of Java, gave overwhelming credit to the legacy of Simula in his lecture. He mentioned that a number of programming languages nowadays compile to JVM. During his lecture, I began to wonder - why has no team made a simula implementation in java, written in java that compiles to java virtual machine(JVM).

Inspired by the lectures on the Simula Jubilee, I began to ask myself if I should try to make a Simula implementation using today's tools. It should certainly be a Recursive Descent parser which I believe is an excellent technique.

In early November I started work on such a compiler. I was surprised that it took so short time to complete a rough version that compiled to Java code. This, first version, was finished just before Christmas. It was not in any way complete, but it generated some kind of Java code that could be compiled and executed. However, the original semantics of the Simula program were not preserved.

I presented these results in a meeting at Blindern, Ole-Johan Dahls Hus, just before Christmas in 2017. I was then encouraged by Stein Krogdahl and Dag Langmyhr to create a full-fledged Simula Implementation strictly after *Simula Standard*

I then established an open source project: Portable Simula Revisited named after the S-Port project in which I had previously worked.

From then on, I worked on a complete redesign of the Simula System. The design is based on previous Simula implementations I have worked on, but this time some major changes have been made. Eg. a separate Garbage Collector is not necessary because the underlying Java system fixes that.

This first Simula System was written in Java and compiles to pure Java source code. The Java compiler was then used to produce bytecode. Then the bytecode is manipulated to reintroduce the goto statement and labels. Finally, the bytecode is executed by the JVM.

A complete implementation was ready in early 2019. After that, my work consisted of rewrite and polishing the code and introducing new Java features. For example VirtualThread which speeds up large simulations.

In the fall of 2022, I became aware of the work on a new API for Class File manipulation. I was wondering if I could use it to generate Java byte code directly without first generating Java Source. I experimented for a while and quickly became convinced that this could be the solution. Then I used the Java code as a template for code generation using the new Java Class File API. This version was completed by Christmas time in 2024.

This document describes the general design of the new Simula System. It is intended as a reference document that should be studied in parallel with the source code of the compiler and its associated runtime system.

The code examples in this document are somewhat simplified. You can use the Compiler option *keepJava* to study the exact code that is generated. You can also use the javap command to study the resulting ClassFiles.

Oslo, january 2025

Øystein Myhre Andersen

Table of Contents

1. The Compiler

This new Simula system is built as a regular recursive descent compiler with an associated runtime system. In addition, it contains an Simula editor; a simple IDE. Both the compiler, the Simula editor and runtime system are written in Java.

The Simula Editor is descibed in the Simula Reference Manual.
The Simula Compiler's javadoc is found here

The compiler consists of the following passes:

- Parsing: Read source file through the scanner building program syntax tree.

- Checking: Traverse the syntax tree performing semantic checking.

- Coding dependent on the CompilerMode:

    - CompilerMode = viaJavaSource:
        - Do JavaCoding: Traverse the syntax tree generating .java code.
        - Call Java Compiler to generate .class files.
        - Do ByteCodeEngineering updating .class files.
        - Create executable .jar of program.
        - Execute .jar file.

    - CompilerMode = directClassFiles:
        - Traverse the syntax tree generating ClassFile code.
        - Create executable .jar of program.
        - Execute .jar file.

    - CompilerMode = simulaClassLoader:
        - Traverse the syntax tree generate and load ClassFile code.
        - Run the loaded program

1.1 The Lexicographic Scanner

The scanner reads the source code and produces a stream of lexical tokens. The following varieties of tokens are produced: Keywords, identifiers, special symbols, numbers, strings and character constants.

In addition, the scanner will treat directive lines; i.e. source input lines starting with the character % Directive lines are described in the Simula Reference Manual.

Tokens basicly consists of three attributes;

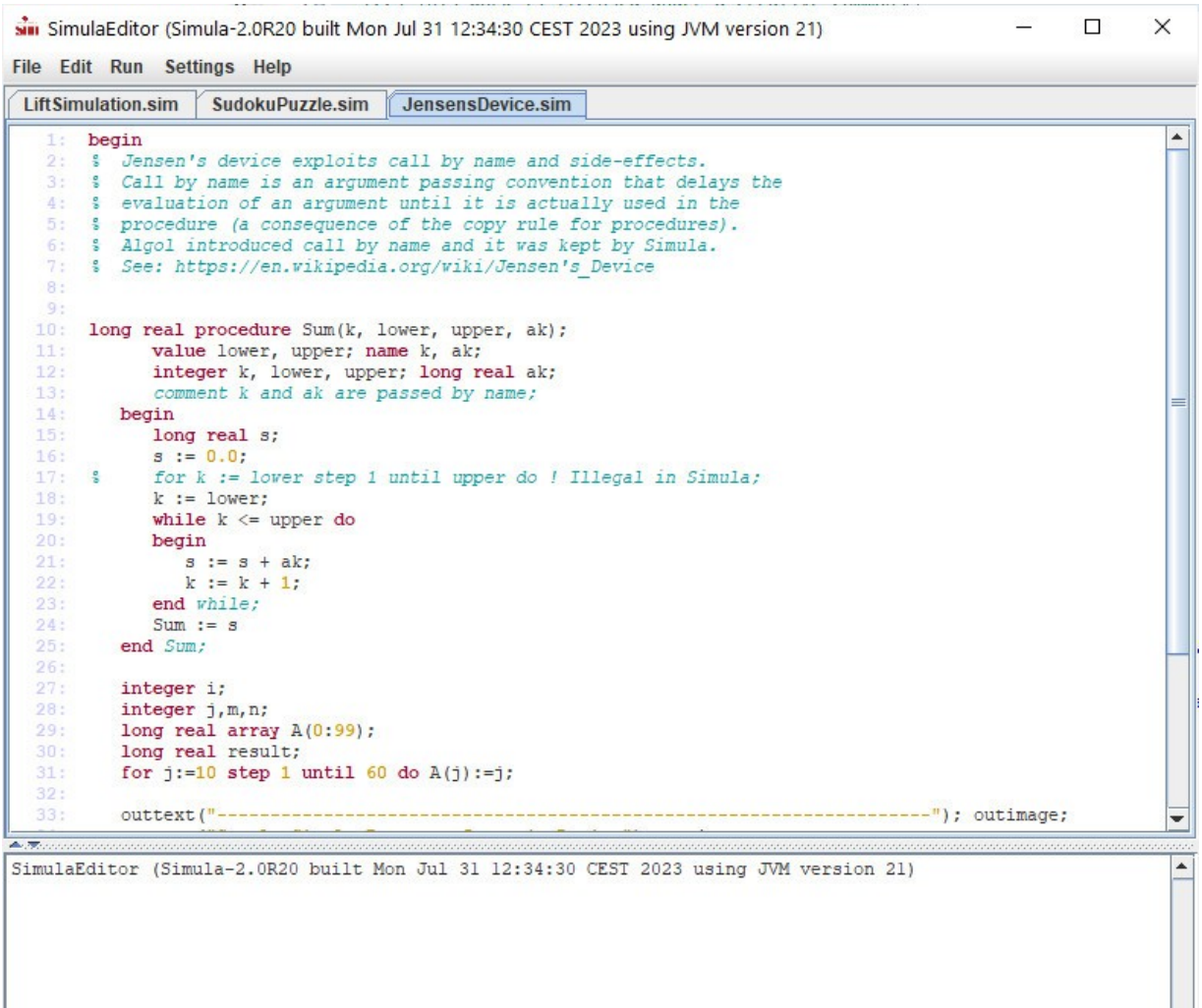
```
class Token {
    KeyWord keyWord; // A Keyword or special symbol
    Object value;    // An attached Object. E.g Number value
    String text;     // The text scanned to form this Token
}
```

The attribute *value* carries identifiers, numbers, strings and character constants.

For further details see the source code of SimulaScanner.java.

The text attribute is used by the Simula Editor. It is displayed in different colors depending upon the KeyWord.

In this way, we achieve context-sensitive editing.



1.2 Parsing.

The compiler uses recursive descent parsing to build a syntax tree. To achieve that, some basic functions are defined:

```
public final class Parse {

    public static Token prevToken;    // The previous Token.
    public static Token currentToken; // The current Token.

    public static void nextToken() {
          prevToken = currentToken;
          Parse.currentToken = simulaScanner.nextToken();
          // ... Test for end-of-file etc.
    }

    public static boolean accept(final KeyWord... key) {
          for (int i = 0; i < key.length; i++)
                if (Parse.currentToken.keyWord == key[i]) {
                      nextToken(); return (true);
                }
          return (false);
    }

    public static boolean expect(final KeyWord key) {
          if (accept(key)) return (true);
          error("Got wrong Token");
          return (false);
    }

}
```

The method *nextToken* will input the next Token from the Scanner and update *prevToken* and *currentToken*.

The method *accept tests currentToken* against each given *key*. If accepted, *nextToken* is called, thus updating *prevToken* and *currentToken*. If no *key* is accepted, false is returned. After a call on *accept* the accepted token is found in *prevToken*.

The method *expect* will call *accept* and if the *key* is accepted, true is returned. However; if not accepted an error message is given and the value false is returned.

In addition to these three basic methods, a number of *accept* and *expect* methods are defined thoughout the implementation. For example *acceptIdentifier* which looks like this:

```
public static String acceptIdentifier() {
      if (Parse.accept(KeyWord.IDENTIFIER))
            return (Parse.prevToken.getIdentifier());
      return (null);
}
```

For further details see the source code of Parse.java.

Parsing a Simula Program

Simula programs are parsed following this syntax:

```
SIMULA-source-module
   = [ external-head ] ( program | procedure-declaration | class-declaration )

   external-head = external-declaration ; { external-declaration ; }

      External-declaration
         = external-procedure-declaration | external-class-declaration

   program = [ block-prefix ] block | [ block-prefix ] compound-statement

      block-prefix = class-identifier [ ( actual-parameter-list ) ]

    procedure-declaration
        = [ type ] PROCEDURE procedure-identifier procedure-head procedure-body
```

Which is coded like this in Java:

```
while(Parse.accept(KeyWord.EXTERNAL)) {
      expectExternalHead( ... );
      Parse.expect(KeyWord.SEMICOLON);
}
String ident=Parse.acceptIdentifier();
if(ident!=null) {
      if(Parse.accept(KeyWord.CLASS))
            module=expectClassDeclaration(ident);
      else {
            Variable blockPrefix=expectVariable(ident);
            Parse.expect(KeyWord.BEGIN);
            module=expectPrefixedBlock(blockPrefix,true);
      }
}
else if(Parse.accept(KeyWord.BEGIN))
     module=createMainProgramBlock();
else if(Parse.accept(KeyWord.CLASS))
     module=expectClassDeclaration(null);
else {
      Type type=Parse.acceptType();
      if(Parse.expect(KeyWord.PROCEDURE))
          module=expectProcedureDeclaration(type);
}
```

Which in turn gives rise to the generation of a syntax tree of the entire program. Each individual node in the tree is an instance of a subclass of *SyntaxClass*.

Many *accept* and *expect* methods have names that end with a non terminal symbol. For example: *expectClassDeclaration*, *expectProcedureDeclaration*, *expectPrefixedBlock*, *acceptDeclaration acceptExpression* etc.

This has been done to improve the understanding of the Java code in terms of the simula syntax in *Simula Standard*.

1.3 The Syntax Tree

The compiler builds a tree consisting of nodes representing some selected non-terminal symbols. Each node is an instance of a subclass of the class SyntaxClass.

The class SyntaxClass has the following general appearance.

```
public abstract class SyntaxClass {

     public void doChecking() {}

     public void doJavaCoding() {}

     public void doDeclarationCoding() {}

     public void buildByteCode(CodeBuilder codeBuilder)
}
```

It contains three virtual methods which is used for semantic checking and Java code generation.

- **doChecking** – Will perform the semantic checking.

Semantic checking is performed by traversing the syntax tree, calling the method *doChecking on all nodes*. It can also be called from other checking code and it is therefore necessary to mark the nodes to avoid repeated calls. It will look like this

```
public void doChecking() {
     if (IS_SEMANTICS_CHECKED()) return;

     … the actuall checking code

     SET_SEMANTICS_CHECKED();
}
```

One of the main tasks during checking is to check the correctness of the source code. An other task is to determine the meaning of all the identifiers.

Code generation is done recursively, starting with the main module. A detailed description is included in section 2. Details of Java code generation

Depending on the compiler mode (Java source or direct class file), these methods will be called by the code generator.

- **doJavaCoding** and **doDeclarationCoding** – will generate Java source code.

- **buildByteCode** – will perform direct class file building using the given CodeBuilder.

1.4 The Syntax Class Hierarchy

The Java class SyntaxClass is the common super class of all Syntax classes. Here is a complete subclass hierarchy of all syntax classes in the compiler.

```
SyntaxClass
   HiddenSpecification
   ProtectedSpecification
   Type
      OverLoad
   Declaration
      ArrayDeclaration
      DeclarationScope
         BlockDeclaration
            ClassDeclaration
               PrefixedBlockDeclaration
               StandardClass
            MaybeBlockDeclaration
            ProcedureDeclaration
               StandardProcedure
               SwitchDeclaration
         ConnectionBlock
      ExternalDeclaration
      Parameter
      SimpleVariableDeclaration
         LabelDeclaration
      VirtualMatch
      VirtualSpecification
   Statement
      ActivationStatement
      BlockStatement
      ConditionalStatement
      ConnectionStatement
      DummyStatement
      ForStatement
      GotoStatement
      InnerStatement
      LabeledStatement
      ProgramModule
      StandaloneExpression
      SwitchStatement
      WhileStatement
   Expression
      ArithmeticExpression
      AssignmentOperation
      BooleanExpression
      ConditionalExpression
      Constant
      LocalObject
      ObjectGenerator
      ObjectRelation
      QualifiedObject
      RelationalOperation
      RemoteVariable
      TextExpression
      TypeConversion
      UnaryOperation
      VariableExpression
```

For further details see the Java packet simula.compiler.syntaxClass.

1.5 Separate Compilation

When the SIMULA-source-module (Simula Standard Chapter 6) is a single class or a procedure, it is considered a separate compilation. In such cases, the compiler will produce .jar files containing both attribute information and a set of .class files. There are such .jar files that are read in by external declarations.

The attribute information is written to an ObjectOutputStrem using Java's externalization technique. Therefore, all syntax classes that can belong to an attribute file must implement the Externalizable interface.

Compatible recompilation is not supported.

2. Details of Java code generation

2.1 Block Instances at runtime.

Simula is a block oriented language. This means that the source text is organized as a set of nested blocks. They can be classes, procedures and sub-blocks. All of which are represented as subclasses of the Java class RTS_RTObject during execution.

The Java class RTS_RTObject has the following main attributes:

`OperationalState STATE`; One of { ***attached,detached,resumed,terminated*** }

boolean isQPSystemBlock()
This is a static property generated by the compiler.
It will return true for Block or Prefixed Block with local classes.

`boolean isDetachable()`
This is a static property generated by the compiler.
It will return true for Classes which can be Detached.

`RTObject SL` This is a pointer to the object of the nearest textually enclosing block instance, also called 'static link'.

`RTObject DL` If this block instance is attached this is a pointer to the object of the block Instance to which the instance is attached (also called dynamic link), i.e. it points to the block instance which called this one.

`Coroutine CORUT` This is a pointer to the Coroutine in which this block instance is running. If this block instance is detached it is used to save the complete reactivation point (call stack and the continuation point).

`int JTX` `Jump Table Index used to implement goto`

`In addition a global variable` ***`CUR`*** `will always` point to the *Current Block*

The figure on next page shows the main structure of nested block instances at runtime.
Ole Johan Dahl had a similar figure in a compendium in 1980. The main differences are:

- There is no Prototype Pointer (we use Java's Object.getClass method instead)
- There is no Context Vector (we follows Static Links SL)
- There is no Garbage Collector (we use Java's instead)

Variables in outer blocks are accessed through the SL-chain. For example:

Variable c in current block	==>	c or ((Block1)CUR).c
Variable b in P2	==>	((P)(CUR.SL)).b
Variable a in outermost block	==>	((Block2)(CUR.SL.SL)).a

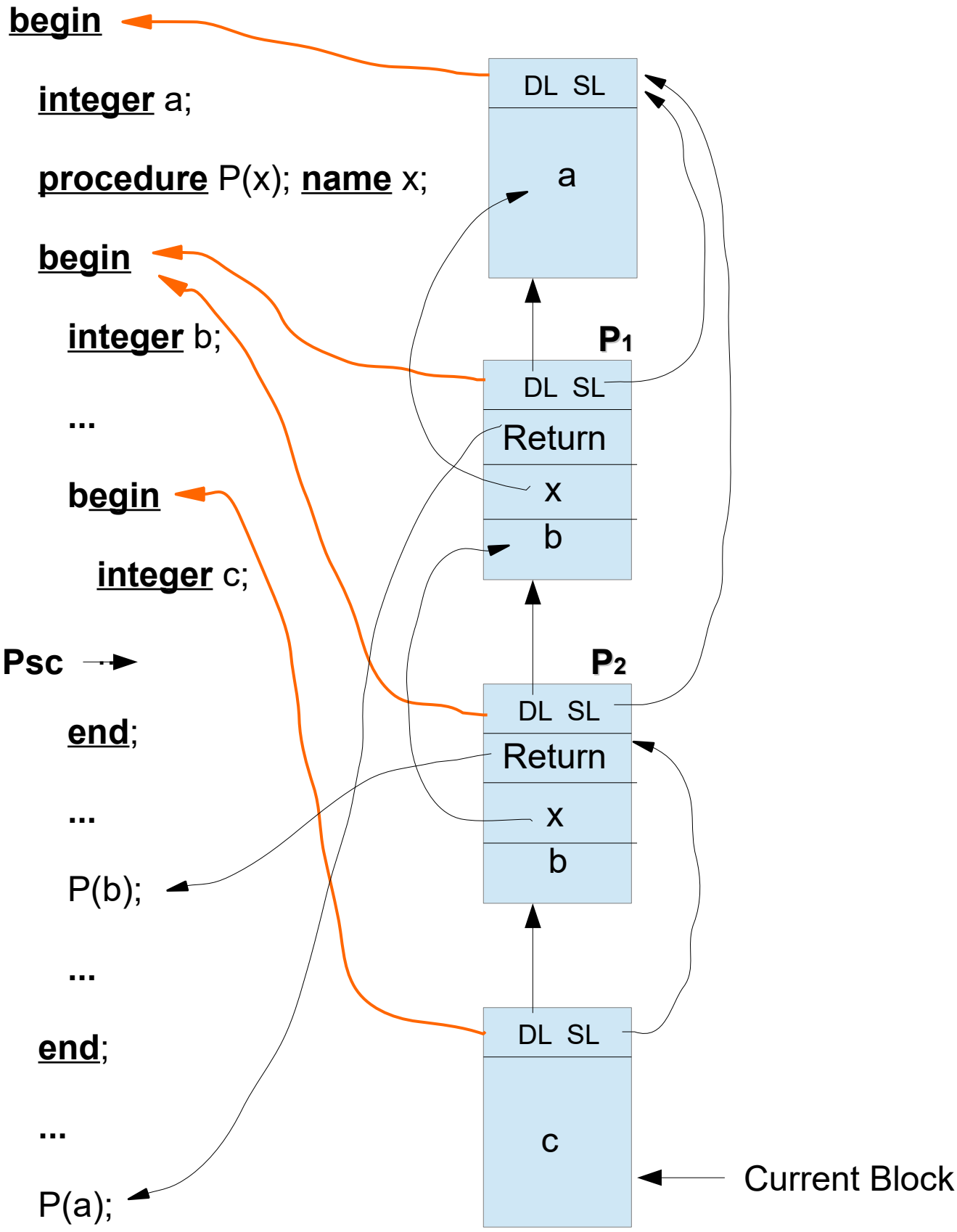


Fig. 1 Sample Block Structure

(due to Ole-Johan Dahl)

2.2 Mapping Simula Blocks to Java classes.

Each and every Sub-Block, Simula Class, Prefixed Block, Procedure or Switch is compiled into a Java Class.

2.2.1 Sub- Block.

A sub-block is the simplest of all Simula Blocks. For example, consider the following Simula code:

```
begin integer lim=40;
      text array TA(1:lim);
      integer i;
      i:=6;
end;
```

It is compiled to a Java class like:

```
public final class SubBlock extends RTS_BASICIO {
    // Declare locals as attributes
    public final int lim=40;
    public RTS_TEXT_ARRAY TA=null;
    public int i=0;
    // Normal Constructor
    public SubBlock2(RTS_RTObject staticLink) {
        super(staticLink);
        BBLK();
        // Declaration Code
        TA=new RTS_TEXT_ARRAY(new RTS_BOUNDS(1,40));
    }
    // SubBlock Statements
    @Override
    public RTS_RTObject STM() {
        i=6;
        EBLK();
        return(this);
    }
}
```

The Java super class BASICIO is defined in the runtime system as a convenient way to enable all environment methods to this block. It is itself a Java sub-class of RTS_RTObject.

The constructor set up the static environment (SL and CORUT) of this RTS_RTObject.

The BBLK method is used to initiate the block instance. It will attach the block to its dynamic environment (DL and CORUT) and update the current block pointer(CUR).
The OperationalState become ***attached***;

The EBLK method is used to terminate the block instance when control passes through the final end (of the outermost prefix for class and prefixed block). This is a complex method dependent on the different kind and states of block instances.
The OperationalState become ***terminated***;

For further details see the source code of MaybeBlockDeclaration.java

Using the ClassFile API, the code building of a Sub-Block will look like this:

```
public byte[] buildClassFile() {

  byte[] bytes = ClassFile.of().build(currentClassDesc(),
    classBuilder -> {
      classBuilder
        .with(SourceFileAttribute.of(Global.sourceFileName))
        .withFlags(ClassFile.ACC_PUBLIC + ClassFile.ACC_FINAL + ClassFile.ACC_SUPER)
        .withSuperclass(RTS.CD.RTS_BASICIO);

      if(this.hasAccumLabel())
        for (LabelDeclaration lab : labelList.getAccumLabels())
          lab.buildDeclaration(classBuilder,this);

      for (Declaration decl : declarationList)
        decl.buildDeclaration(classBuilder,this);

      if (isQPSystemBlock())
        classBuilder
         .withMethodBody("isQPSystemBlock", MethodTypeDesc.ofDescriptor("()Z"),
            ClassFile.ACC_PUBLIC, codeBuilder -> buildIsQPSystemBlock(codeBuilder));

      classBuilder
        .withMethodBody("<init>",
           MethodTypeDesc.ofDescriptor("(Lsimula/runtime/RTS_RTObject;)V"),
           ClassFile.ACC_PUBLIC,
           codeBuilder -> buildConstructor(codeBuilder))
        .withMethodBody("_STM",
           MethodTypeDesc.ofDescriptor("()Lsimula/runtime/RTS_RTObject;"),
           ClassFile.ACC_PUBLIC,
           codeBuilder -> buildMethod_STM(codeBuilder));

      if (this.isMainModule)
        classBuilder
          .withMethodBody("main",
            MethodTypeDesc.ofDescriptor("([Ljava/lang/String;)V"),
            ClassFile.ACC_PUBLIC + ClassFile.ACC_STATIC + ClassFile.ACC_VARARGS,
              codeBuilder -> buildMethodMain(codeBuilder));
    }
  );
  return(bytes);
}
```

The byte array can then be written to a .class file.

For further details see the source code of MaybeBlockDeclaration.java

2.2.2 Simula Classes

Consider the following Simula Class:

```
Class A(p1,p2); integer p1; real p2;
begin integer lim=40;
      text array TA(1:lim);
      integer i; real r;
      i:=p1;
      inner;
      r:=p2;
 end;
```

It will be compiled to a Java Class like:

```
public class A extends RTS_CLASS {
    // Declare parameters as attributes
    public int p_p1;
    public float p_p2;
    // Declare locals as attributes
    public final int lim=(int)(40);
    public RTS_TEXT_ARRAY TA=null;
    public int i=0;
    public float r=0.0f;
    // Normal Constructor
    public A(RTS_RTObject staticLink,int sp_p1,float sp_p2) {
        super(staticLink);
        // Parameter assignment to locals
        this.p_p1 = sp_p1;
        this.p_p2 = sp_p2;
        BBLK(); // Iff no prefix
        // Declaration Code
        TA=new RTS_TEXT_ARRAY(new RTS_BOUNDS(1,40));
    }
    // Class Statements
    @Override
    public A STM() {
        // Class A: Code before inner
        i=p_p1;
        // Class A: Code after inner
        r=p_p2;
        EBLK();
        return(this);
    }
}
```

Here we see that both parameters and local variables are declared as local members. The constructor consists of parameter transmission and Array declaration code.

The member method STM contains the concatenated statement code belonging to this Simula Class.

RTS_CLASS is a subclass of RTS_RTObject which define the Detach method.

For further details see the source code of ClassDeclaration.java

2.2.3 Sub-classes

Now let's look at a sub-class of Simula Class A:

```
A Class B(p3); long real p3;
begin
      array RA(1:lim);
      RA(14):=r;
      inner;
      r:=RA(14);
end;
```

Which will be compiled to a Java Class like:

```
public class B extends A {
    // Declare parameters as attributes
    public double p1_p3;
    // Declare locals as attributes
    public RTS_REAL_ARRAY RA=null;
    // Normal Constructor
    public B(RTS_RTObject staticLink,int sp_p1,float sp_p2,double sp1_p3) {
        super(staticLink,sp_p1,sp_p2);
        // Parameter assignment to locals
        this.p1_p3 = sp1_p3;
        // Declaration Code
        RA=new RTS_REAL_ARRAY(new RTS_BOUNDS(1,40));
    }
    // Class Statements
    @Override
    public B STM() {
        // Class A: Code before inner
        i=p_p1;
        // Class B: Code before inner
        RA.putELEMENT(RA.index(14),r);
        // Class B: Code after inner
        r=RA.getELEMENT(14);
        // Class A: Code after inner
        r=p_p2;
        EBLK();
        return(this);
    }
}
```

Here we see that:

- Parameters are accumulated via the Constructors.

- Declaration code concatenation is done via the constructors.

- Concatenation of Statement code is done explicitly by copying the Java code I.e. 'inner' is optimized by textual concatenation.

Object generation, i.e. new B(1,2,3) is compiled to:

```
x=new B(CUR,1,2f,3d).STM();
```

Where *CUR* is a global variable pointing at the 'Current Block'

For further details see the source code of ClassDeclaration.java.

Using the ClassFile API, the code building of a Class will look like this:

```
public byte[] buildClassFile() {


  byte[] bytes = ClassFile.of().build(currentClassDesc(),
    classBuilder -> {
      ClassBuilder
        .with(SourceFileAttribute.of(Global.sourceFileName))
        .withFlags(ClassFile.ACC_PUBLIC + ClassFile.ACC_SUPER)
        .withSuperclass(superClassDesc());

      if(this.hasAccumLabel())
        for (LabelDeclaration lab : labelList.getAccumLabels())
          lab.buildDeclaration(classBuilder,this);

      for (Declaration decl : declarationList)
        decl.buildDeclaration(classBuilder,this);

      for (Parameter par : parameterList)
        par.buildDeclaration(classBuilder,this);

      for (VirtualSpecification virtual : virtualSpecList)
        if (!virtual.hasDefaultMatch)
          virtual.buildMethod(classBuilder);

      for (VirtualMatch match : virtualMatchList)
        match.buildMethod(classBuilder);

      classBuilder
        .withMethodBody("<init>",
          MethodTypeDesc.ofDescriptor(edConstructorSignature()),
          ClassFile.ACC_PUBLIC,
            codeBuilder -> buildConstructor(codeBuilder))
        .withMethodBody("_STM",
          MethodTypeDesc.ofDescriptor("()Lsimula/runtime/RTS_RTObject;"),
          ClassFile.ACC_PUBLIC,
            codeBuilder -> buildMethod_STM(codeBuilder));

      if (isDetachUsed())
        classBuilder
          .withMethodBody("isDetachUsed",
            MethodTypeDesc.ofDescriptor("()Z"),
            ClassFile.ACC_PUBLIC,
              codeBuilder -> buildIsMethodDetachUsed(codeBuilder));
   }
  );
  return(bytes);
}
```

The byte array can then be written to a .class file.

For further details see the source code of ClassDeclaration.java

2.2.4 Prefixed Block.

A prefixed block is very much like a sub-class:

```
B(1,2,3) begin
begin
      text array TA(1:lim);
      integer i;
      i:=6;
end;
```

Will be compiled to:

```
new PBLK(CUR,1,2f,3d).STM();
```

Where PBLK is a Java Class like:

```
public final class PBLK extends B {
    // Declare locals as attributes
    public RTS_TEXT_ARRAY TA=null;
    public int i_2=0;
    // Normal Constructor
    public PBLK(RTS_RTObject staticLink,int sp_p1,float sp_p2,double sp1_p3) {
        super(staticLink,sp_p1,sp_p2,sp1_p3);
        // Declaration Code
        TA=new RTS_TEXT_ARRAY(new RTS_BOUNDS(1,40));
    }
    // Class Statements
    @Override
    public PBLK STM() {
        // Class A: Code before inner
        i=p_p1;
        // Class B: Code before inner
        RA.putELEMENT(RA.index(14),r);
        // PBLK: Code
        i_2=6;
        // Class B: Code after inner
        r=RA.getELEMENT(14);
        // Class A: Code after inner
        r=p_p2;
        EBLK();
        return(this);
    }
}
```

Again we see that:

- Parameters are accumulated via the Constructors.

- Declaration code concatenation is done via the Constructors.

- Concatenation of Statement code is done explicitly by copying the Java code

For further details see the source code of PrefixedBlockDeclaration.java.

Using ClassFile API, the code building of a Prefixed Block will look like this:

```
public byte[] buildClassFile() {


  byte[] bytes = ClassFile.of().build(currentClassDesc(),
    classBuilder -> {
      classBuilder
        .with(SourceFileAttribute.of(Global.sourceFileName))
        .withFlags(ClassFile.ACC_PUBLIC + ClassFile.ACC_SUPER)
        .withSuperclass(this.superClassDesc());

      if(this.hasAccumLabel())
        for (LabelDeclaration lab : labelList.getAccumLabels())
          lab.buildDeclaration(classBuilder,this);

      for (Declaration decl : declarationList)
        decl.buildDeclaration(classBuilder,this);

      for(Parameter par:parameterList)
        par.buildDeclaration(classBuilder,this);

      for (VirtualSpecification virtual : virtualSpecList)
        if (!virtual.hasDefaultMatch)
          virtual.buildMethod(classBuilder);

      for (VirtualMatch match : virtualMatchList)
        match.buildMethod(classBuilder);

      classBuilder
        .withMethodBody("<init>",
          MethodTypeDesc.ofDescriptor(edConstructorSignature()),
          ClassFile.ACC_PUBLIC,
          codeBuilder -> buildConstructor(codeBuilder))
        .withMethodBody("_STM",
          MethodTypeDesc.ofDescriptor("()Lsimula/runtime/RTS_RTObject;"),
          ClassFile.ACC_PUBLIC,
          codeBuilder -> buildMethod_STM(codeBuilder) );

      if (isQPSystemBlock())
        classBuilder.withMethodBody("isQPSystemBlock",
          MethodTypeDesc.ofDescriptor("()Z"), ClassFile.ACC_PUBLIC,
          codeBuilder -> buildIsQPSystemBlock(codeBuilder));

      if (isDetachUsed())
        classBuilder.withMethodBody("isDetachUsed",
          MethodTypeDesc.ofDescriptor("()Z"), ClassFile.ACC_PUBLIC,
          codeBuilder -> buildIsMethodDetachUsed(codeBuilder));

      if (this.isMainModule)
        classBuilder.withMethodBody("main",
          MethodTypeDesc.ofDescriptor("([Ljava/lang/String;)V"),
          ClassFile.ACC_PUBLIC + ClassFile.ACC_STATIC + ClassFile.ACC_VARARGS,
          codeBuilder -> buildMethodMain(codeBuilder));
    }
  );
  return(bytes);
}
```

The byte array can then be written to a .class file.

For further details see the source code of PrefixedBlockDeclaration.java.

2.2.5 Procedure Declaration

Procedures however; follow a different scheme.

```
integer Procedure P(p1,p2); integer p1; real p2;
begin text Array TA(1:40); integer i;
      P:=p1;
      i:=6;
end;
```

No concatenation in this case but the procedure must be prepared to be called as a formal procedure without parameter specifications.

```
public final class P extends RTS_PROCEDURE {
    // Declare return value as attribute
    public int RESULT;
    @Override
    public Object RESULT() { return(RESULT); }
    // Declare parameters as attributes
    public int p_p1;
    public float p_p2;
    // Declare locals as attributes
    public RTS_TEXT_ARRAY TA=null;
    public int i=0;
    // Parameter Transmission in case of Formal/Virtual Procedure Call
    @Override
    public P setPar(Object param) {
        try { switch(nParLeft--) {
                case 2: p_p1=intValue(param); break;
                case 1: p_p2=floatValue(param); break;
                default: throw new RTS_SimulaRuntimeError("Too many parameters");
            }
        } catch(ClassCastException e) {
             throw new RTS_SimulaRuntimeError("Wrong type of parameter: "+param,e); }
        return(this);
    }
    // Constructor in case of Formal/Virtual Procedure Call
    public P(RTS_RTObject SL) {
        super(SL,2); // Expecting 2 parameters
    }
    // Normal Constructor
    public P(RTS_RTObject SL,int sp_p1,float sp_p2) {
        super(SL);
        // Parameter assignment to locals
        this.p_p1 = sp_p1;
        this.p_p2 = sp_p2;
        BBLK();
        // Declaration Code
        TA=new RTS_TEXT_ARRAY(new RTS_BOUNDS(1,40));
        STM();
    }
    // Procedure Statements
    @Override
    public P STM() {
        RESULT=p_p1;
        i=6;
        EBLK();
        return(this);
    }
}
```

For further details see the source code of ProcedureDeclaration.java.

Using the ClassFile API, the code building a Procedure will look like this:

```
public byte[] buildClassFile() {


  byte[] bytes = ClassFile.of().build(currentClassDesc(),
    ClassBuilder -> {
      classBuilder
        .with(SourceFileAttribute.of(Global.sourceFileName))
        .withFlags(ClassFile.ACC_PUBLIC + ClassFile.ACC_FINAL + ClassFile.ACC_SUPER)
        .withSuperclass(RTS.CD.RTS_PROCEDURE);

      if(type != null)
        classBuilder
          .withMethodBody("_RESULT",
            MethodTypeDesc.ofDescriptor("()Ljava/lang/Object;"),
            ClassFile.ACC_PUBLIC,
            codeBuilder -> buildMethod_RESULT(codeBuilder));

      for (Parameter par:parameterList)
        par.buildDeclaration(classBuilder,this);

      if(this.hasAccumLabel())
        for (LabelDeclaration lab : labelList.getAccumLabels())
          lab.buildDeclaration(classBuilder,this);

      for (Declaration decl : declarationList)
        decl.buildDeclaration(classBuilder,this);

      if(parameterList.size() > 0)
        classBuilder
         .withMethod("setPar",
            MethodTypeDesc.ofDescriptor(
              "(Ljava/lang/Object;)Lsimula/runtime/RTS_PROCEDURE;"),
            ClassFile.ACC_PUBLIC,
            codeBuilder -> buildSetPar(codeBuilder))
         .withMethod("<init>", MTD_Constructor(false), ClassFile.ACC_PUBLIC,
            codeBuilder -> buildConstructor2(codeBuilder));

      classBuilder
        .withMethod("<init>", MTD_Constructor(true), ClassFile.ACC_PUBLIC,
          codeBuilder -> buildConstructor(codeBuilder))
        .withMethodBody("_STM",
          MethodTypeDesc.ofDescriptor("()Lsimula/runtime/RTS_RTObject;"),
          ClassFile.ACC_PUBLIC,
          codeBuilder -> buildMethod_STM(codeBuilder));
    }
  );
  return(bytes);
}
```

The byte array can then be written to a .class file.

For further details see the source code of ProcedureDeclaration.java.

2.2.5.1 Procedure Calls

Ordinary procedure-call are similar to class object generation:

```
P(4,3.14);          ==>   new P(CUR,4,3.14f);
x.P(4,3.14);        ==>   new P(x,4,3.14f);
Inspect x do P(4,3.14);      ==>   if(x!=null) new P(x,4,3.14f);
```

However, we need to pick up the result of <type>procedures.

```
i:=P(4,3.14);     ==>   i=new P(CUR,4,3.14f).RESULT;
i:=x.P(4,3.14);   ==>   i=new P(x,4,3.14f).RESULT;
Inspect x do i:=P(4,3.14);   ==>   if(x!=null) i=new P(x,4,3.14f).RESULT;
```

Formal Procedures

A call on a formal Procedure; however; will look quite different. Suppose that F is a procedure quantity representing a formal parameter procedure.
Then a call like `i:=F(7,9);` will generate code like this:

```
i = intValue(F.CPF()
       .setPar(new RTS_NAME<Integer>() { public Integer get() { return (7); }})
       .setPar(new RTS_NAME<Integer>() { public Integer get() { return (9); }})
       .ENT().RESULT());
```

The parameters are transmitted by repeated calls on the compiler-generated method 'setpar'. Since we neither know the transfer mode nor the type, the parameter is transferred 'by name' and post-processed by 'setpar'.

Virtual Procedures

A Simula virtual procedure is represented by a Java virtual member method in the Java class. The method returns a Procedure Quantity (PRCQNT) pointing to the matching Simula Procedure Declaration. We use Java's virtual concept to implement Simula's virtual procedures. When a procedure redefinition occur in a subclass, an additional member method is generated by the compiler.

If there is no matching procedure a default method is generated by the compiler:

```
public PRCQNT V(){ throw new RuntimeException("No Virtual Match: V"); }
```

If a matching procedure is declared the following method is generated:

```
public PRCQNT V() { return(new PRCQNT(this,V.class)); }
```

In this way, we ensure that the correct virtual match is called.

A call on a virtual Procedure uses the V() method to select the virtual match. Apart from that a call on a virtual procedure will follow the same pattern as formal procedures. However, if the virtual procedure is specified, a call like `i:=x.V(4,3.14);`
will generate this simplified code:

```
i=intValue(x.V().CPF().setPar(4).setPar(3.14f).ENT().RESULT());
```

In this case transmission 'by name' is not necessary.

For further details see the source code of PRCQNT.java..

2.2.6 Switch Declaration

Switch declarations are treated as LABEL Procedures without any optimization whatsoever.

Consider the following example:

```
switch S2:=L1,L2,S(3),S( expr );
```

This code will be generated:

```
public final class S2 extends RTS_PROCEDURE {
    // Declare return value as attribute
    public RTS_LABEL RESULT;
    @Override
    public Object RESULT() { return(RESULT); }
    // Declare parameters as attributes
    public int p_SW;
    // Parameter Transmission in case of Formal/Virtual Procedure Call
    @Override
    public S2 setPar(Object param) {
        try {
            switch(nParLeft--) {
                case 1: p_SW=intValue(param); break;
                default: throw new RTS_SimulaRuntimeError("Too many parameters");
            }
        } catch(ClassCastException e) {
            throw new RTS_SimulaRuntimeError("Wrong type of parameter: "+param,e);
        }
        return(this);
    }
    // Constructor in case of Formal/Virtual Procedure Call
    public S2(RTS_RTObject SL) {
        super(SL,1); // Expecting 1 parameters
    }
    // Normal Constructor
    public S2(RTS_RTObject SL,int sp_SW) {
        super(SL);
        // Parameter assignment to locals
        this.p_SW = sp_SW;
        BBLK();
        // Declaration Code
        STM();
    }
    // Switch Body
    @Override
    public S2 STM() {
        switch(p_SW-1) {
            case 0: RESULT=(( encl )(CUR.SL)).LABEL_L1; break;
            case 1: RESULT=(( encl )(CUR.SL)).LABEL_L2; break;
            case 2: RESULT=new S((CUR.SL),3).RESULT; break;
            case 3: RESULT=new S((CUR.SL),( expr ).RESULT; break;
            default: throw new RTS_SimulaRuntimeError("Illegal switch index”);
        }
        EBLK();
        return(this);
    }
}
```

For further details see the source code of SwitchDeclaration.

2.2.6.1 Switch Designators

Ordinary switch designators are similar to call on procedures.

```
Goto S(4);   ==>   throw new S((CUR),4).RESULT;
```

*Switches are implicit specified **protected**, thus goto a remote or connected switch is illegal.*

Formal Switches

Formal switches follows the pattern of procedures.

Suppose that F is a procedure quantity representing a formal switch.
Then the statement: goto *F(expr);* will generate code like this:

```
GOTO((RTS_LABEL)(F.CPF()
   .setPar(new RTS_NAME<Integer>(){ public Integer get() { return( expr ); }})
   .ENT().RESULT()));
```

Virtual Switches

A virtual switch follows the pattern of virtual procedures. It is represented by a Java virtual member method in the Java class. The method returns a Procedure Quantity (PRCQNT) pointing to the matching Simula Switch Declaration.

2.3 Representation of Quantities.

Within an object, each quantity local to that object is represented. Local quantities are e.g. Declared variables in a block (sub-block, procedure body or prefixed block), or the attribute of a class. Quantities may be of value types, object reference type, text descriptors, array quantities, procedure quantities, label quantities or switch quantities..

2.3.1 Value type variables.

A variable of value type is represented in Java as follows:

Boolean	boolean
Character	char
Short integer	int *)
Integer	int
Real	float
Long real	double

*) Simula Standard 2.1.1 states; " An implementation may choose to implement short integer exactly as integer, i.e. ignoring the keyword short "

2.3. 2 Object Reference Types.

Object reference variables are represented in Java as such.

ref(ClassIdentifier)	ClassIdentifier
none	**null**

2.3. 3 Text Reference Types.

Text reference variables is a composite structure represented by a RTS_TXT Java Object.

Text	RTS_TXT

where RTS_TXT has the following definition:

```
public class RTS_TXT extends RTS_RTObject {
   RTS_TEXTOBJ OBJ; // Reference to the text object.
   int START;       // Start index of OBJ.MAIN[], counting from zero.
                    // Note this differ from Simula Definition.
   int LENGTH;
   int POS;         // Current index of OBJ.MAIN[], counting from zero.
                    // Note this differ from Simula Definition.
   ...
}
```

The special text constant **notext** is represented by Java **null**.

The object RTS_TEXTOBJ is defined as:

```
public class RTS_TEXTOBJ extends RTS_RTObject {
   int SIZE;       // Number of characters in the text object.
   boolean CONST;  // True: Indicates a text constant
   char[] MAIN;    // The characters
   ...
}
```

2.3.4 Array Quantities.

A local array is represented by a reference to an array object which is a subclass of RTS_ARRAY:

```
public abstract class RTS_ARRAY {
	public  final RTS_BOUNDS[] BOUNDS;
	public  final int SIZE;
	private final int BASE;
	final private int[] DOPE;

	public RTS_ARRAY(final RTS_BOUNDS... BOUNDS) {
		this.BOUNDS = BOUNDS;
		DOPE=new int[BOUNDS.length];
		DOPE[0]=1;
		int SIZE=BOUNDS[0].SIZE;
		int BASE=BOUNDS[0].LB * DOPE[0];
		for(int i=1;i<BOUNDS.length;i++) {
			DOPE[i] = BOUNDS[i-1].SIZE * DOPE[i-1];
			BASE=BASE + BOUNDS[i].LB * DOPE[i];
			SIZE=SIZE * BOUNDS[i].SIZE; }
		this.SIZE=SIZE;
		this.BASE=BASE;
	}

	public int nDim() { return (BOUNDS.length); }
	public int size() { return (SIZE); }
	public int lowerBound(int i) { return(BOUNDS[i].LB); }
	public int upperBound(int i) { return(BOUNDS[i].LB + BOUNDS[i].SIZE - 1); }

	public abstract RTS_ARRAY COPY();

	public int index(int... x) {
		int idx=0;
		for(int k=0;k<x.length;k++) {
			if(x[k] < lowerBound(k) || x[k] > upperBound(k))
				throw new _SimulaRuntimeError("Array index("+(k+1)+") = "+x[k]
					+" is outside bounds "+BOUNDS[k]);
			idx=idx+(x[k] * DOPE[k]);
		}
		return(idx - BASE);
	}
}
```

The implementation technique used is called 'dope vector indexing'. The dope vector is calculated in the constructor based on the array bound pairs. A fictitious base is also Calculated. This base is the relative address of the array element (0,0, ... 0).

Indexing is done by using this data in the 'index' method.

A parameter Array is represented by the RTS_ARRAY quantity.

For further details see the source code of RTS_ARRAY.java.

Arrays of different types are declared [type] array, for example:

```
public final class RTS_INT_ARRAY extends RTS_ARRAY {
      final private int[] ELTS;
      public RTS_INT_ARRAY(final RTS_BOUNDS... BOUNDS) {
            super(BOUNDS); ELTS=new int[SIZE]; }
      public int putELEMENT(int ix,int val) {
            ELTS[ix]=val; return(val); }
      public int getELEMENT(int... x) {
            return(ELTS[index(x)]); }
      public RTS_INT_ARRAY COPY() {
            RTS_INT_ARRAY copy = new RTS_INT_ARRAY(BOUNDS);
            System.arraycopy(ELTS, 0, copy.ELTS, 0, SIZE);
            return(copy); }
}
```

Example: A one-dimensional *Integer Array Table(6:56)* is declared by:

```
table=new RTS_INT_ARRAY(new RTS_BOUNDS(6,56));
```

And we access elements like this:

```
i=table.getELEMENT(x);
table.putELEMENT(table.index(x),i);
```

And, a muliti-dimesional t*ext Array TA(6:56,13:34,-74:-30, ...)* is coded like:

```
ta=new RTS_TEXT_ARRAY(new RTS_BOUNDS(6,56),new RTS_BOUNDS(13,34)
                      ,new RTS_BOUNDS(-74,-30), ...);
```

And we access elements like this:

```
t=ta.getELEMENT(45,21,-60, ...);
ta.putELEMENT(ta.index(45,21,-60, ...),t);
```

For further details see the source code of RTS_INT_ARRAY.java.

2.3.5 Procedure Quantities.

A procedure is represented as a composite structure as shown below. Procedure quantities are used to represent formal procedure parameters. Local procedures, however, are not represented as quantities within an object.

```
public class RTS_PRCQNT {
  RTS_RTObject staticLink;
  Class<?> procedure; // The Java class representing the Simula Procedure.

  // Constructor
  public RTS_PRCQNT(RTS_RTObject staticLink,Class<?> procedure)
  { this.staticLink=staticLink; this.procedure=procedure; }

  public RTS_PROCEDURE CPF() {
    Try {
      Constructor<?> constr = procedure.getConstructor(new Class[]{RTS_RTObject.class});
      Object obj=constr.newInstance(staticLink);
      return((RTS_PROCEDURE)obj);
    }
    catch(Throwable e) { throw new RuntimeException("Internal Error",e);     }
  }

}
```

The attribute 'staticLink' is a pointer to the block-instance in which the procedure is declared.

In traditional Simula implementations, a prototype is generated to hold basic informations of block-instances, including procedures. In this implementation no Prototype is generated. Instead, we use Java's reflection mechanisms to create procedure objects.

The attribute 'procedure' is a pointer to Java's 'prototype' i.e. a pointer to a Java Object containing all information of the class representing the Simula procedure.

The CPF method is used to create procedure object when generating call to formal or virtual procedures. Further details in a later section.

Suppose we have a Simula procedure 'P'. To create a procedure quantity at runtime we use constructs like this:

```
new RTS_PRCQNT(CUR,P.class);     // P is local
new RTS_PRCQNT(CUR.SL,P.class); // P is declared at an outer block
```

For further details see the source code of RTS_PRCQNT.java.

2.3.6 Label Quantities.

Label quantities are explained in chp. 6.1.1.

2.3.7 Switch Quantities.

This implementation treats a Simula Switch as a RTS_LABEL Procedure.

2.4 Parameter Transmission.

2.4.1 Parameter Transmission 'by Name'.

The basic principle is to establish an object within the calling scope. This object will have two attributt methods; 'get' og 'put' to read the value of the actual parameter, or, if legal, to write into it. The following Java-class is used to perform such parameter transmissions:

```
abstract class RTS_NAME<T> {
      abstract T get();
      void put(T x) { error("Illegal ..."); }
}
```

Note that we both use abstract Java classes and 'generics' i.e. the actual type is a parameter. Also note that the 'put' method has a default definition producing an error. This enables redefinition of the 'put' method to be dropped for expression as actual parameters.

Suppose the Simula Procedure:

***procedure** P(k); **name** k; **integer** k; k:=k+1;*

It will be translated to something like this Java method:

```
void P(RTS_NAME<Integer> k) {
      k.put(k.get() + 1); // E.g: k=k+1
}
```

In the calling place, in practice in the actual parameter list, we create an object of a specific subclass of RTS_NAME<*T*> by specifying the Integer type and defining the get and put methods. Eg. If the current parameter is a variable 'q', then the actual parameter will be coded as follows:

```
 new RTS_NAME<Integer>() {
      Integer get() { return (q); }
      void put(Integer x) { q = (int) x;  }
}
```

However, if the actual parameter is an expression like (j + m * n) then it will be coded as follows:

```
new RTS_NAME<Integer>() {
      Integer get() { return (j + m * n);       }
}
```

Here we see that the 'put' method is not redefined so that any attempt to assign a new value to this name parameter will result in an error message.

A classic example of name parameters is found in Wikipedia's article about *Jensen's Device*. We change it slightly because Simula does not allow the control variable in a for-statement to be a formal parameter transferred by name.

```
long real procedure sum(k, lower, upper, ak);
    value lower, upper; name k, ak;
integer k, lower, upper; long real ak;
    begin long real s;
        s := 0.0;
        k := lower;
        while k <= upper do
        begin
            s := s + ak;
            k := k + 1;
        end while;
        sum := s
    end;
```

Translated to Java, this will be:

```
public final class sum extends RTS_PROCEDURE {
    // Declare return value as attribute
    public double RESULT;
    @Override
    public Object RESULT() { return(RESULT); }
    // Declare parameters as attributes
    public RTS_NAME<Integer> k;
    public int lower;
    public int upper;
    public RTS_NAME<Double> ak;
    // Declare locals as attributes
    public double s=0.0d;
    // Normal Constructor
    public sum(RTS_RTObject SL,
               RTS_NAME<Integer> k,int lower,int upper,RTS_NAME<Double> ak) {
        super(SL);
        // Parameter assignment to locals
        this.k = k;
        this.lower = lower;
        this.upper = upper;
        this.ak = ak;
        BBLK();
        // Declaration Code
        STM();
    }
    // Procedure Statements
    @Override
    public Sum STM() {
        s=0.0d;
        k.put(lower);
        while(k.get() <= upper) {
                s=s+(ak.get());
                k.put(k.get()+1);
        }
        RESULT=s;
        EBLK();
        return(this);
    }
}
```

A procedure call like this:

```
integer i;
long real array A(0:99);
long real result;

resultat=sum(i,10,60,A(i));
```

Is translated to:

```
public final class JensensDevice extends RTS_BASICIO {
    // Declare locals as attributes
    public int i=0;
    public RTS_LONG_REAL_ARRAY A=null;
    public double result=0.0d;
    // Normal Constructor
    public JensensDevice(RTS_RTObject staticLink) {
        super(staticLink);
        BBLK();
        BPRG("JensensDevice");
        // Declaration Code
        A=new RTS_LONG_REAL_ARRAY(new RTS_BOUNDS(0,99));
    }
    // SimulaProgram Statements
    @Override
    public RTS_RTObject STM() {
      result= new JensensDevice_Sum(CUR,
                  new RTS_NAME<Integer>(){
                        public Integer get() { return(i); }
                        public Integer put(Integer x) { return(i=(int)x); }
                  },
                  10,
                  60,
                  new RTS_NAME<Double>(){
                        public Double get() { return(A.getELEMENT(i)); }
                        public Double put(Double x) {
                              return(A.putELEMENT(A.index(i),x)); }
                  }).RESULT;
        ;
        EBLK();
        return(this);
    }
}
```

2.4.2 Array as actual parameter

A procedure call with array parameter *i:=P(A);* where the parameter A is specified 'value' will generate code like:

```
i=new P(CUR,A.COPY()).RESULT;
```

Where the method 'COPY' is defined in the runtime system.

When the parameter is specified 'name' and the actual parameter expression is:
i:=P(if cond then A else B); *the following code is* generated:

```
i=new P(CUR,new RTS_NAME<RTS_ARRAY>(){
     public RTS_ARRAY get() { return(cond?A:B); } }).RESULT;
```

And, finally, in default transmission mode the `code will be:`

```
i=new P(CUR,A).RESULT;
```

In all cases it is assumed that the procedure P is declared in the same block as the procedure call. If not, substitute '*CUR*' with '*CUR.SL...*' to refer the static environment of the procedure.

2.4.3 Procedure identifier as actual parameter to formal procedure

In this case, we don't know if it is the name or value P to be transferred. In all cases the parameter is represented by a `NAME$<PRCQNT$>` which can be used to obtain a reference to the actual parameter procedure. Which in turn can be used to evaluate the value of P.

ActualParameter is visible procedure

A procedure call with procedure parameter *i:=Q(P);* where the parameter P is specified 'by name' and the actual parameter is an ordinary visible procedure will generate code like:

```
i=new Q(_CUR, new RTS_NAME<RTS_PRCQNT>() {
       public RTS_PRCQNT get() { return(new RTS_PRCQNT(CUR, P.class)); }
}).RESULT;
```

ActualParameter is remote procedure

A procedure call with procedure parameter *i:=Q(x.P);* where the parameter is specified 'by name' and the actual parameter is an remote procedure will generate code like:

```
i=new Q(CUR,new RTS_NAME<RTS_PRCQNT>() {
       public RTS_PRCQNT get() { return(new RTS_PRCQNT(x, P.class)); }
}).RESULT;
```

ActualParameter is connected procedure

A procedure call with procedure parameter *inspect x do r := Q(P);* where the parameter is specified 'by name' and the actual parameter is an connected procedure will generate code like:

```
if (x != null) // INSPECT x
      i = new Q(CUR, new RTS_NAME<RTS_PRCQNT>() {
            public RTS_PRCQNT get() { return(new RTS_PRCQNT(x, P.class); }
      }).RESULT;
```

ActualParameter is virtual procedure

A procedure call with procedure parameter *i:=Q(V);* where the parameter V is specified 'by name' and the actual parameter is an virtual procedure will generate code like:

```
i=new Q(CUR.SL … ,new RTS_NAME<RTS_PRCQNT>() {
        public RTS_PRCQNT get() { return(V()); }
  }).RESULT;
```

Where V() is the RTS_PRCQNT get() method used to evaluate the virtual match.
See also 5.1.5 Virtual Procedures.

ActualParameter is remote virtual procedure

A procedure call with procedure parameter *i:=Q(x.V);* where the parameter V is specified 'by name' and the actual parameter is a remote virtual procedure will Generate code like:

```
i = new Q(CUR, new RTS_NAME<RTS_PRCQNT>() {
      public RTS_PRCQNT get() { return (x.V()); }
}).RESULT;
```

Where V() is the RTS_PRCQNT get() method used to evaluate the virtual match.

ActualParameter is connected virtual procedure

A procedure call with procedure parameter *inspect x do r := Q(V);* where the parameter V is specified 'by name' and the actual parameter is a connected virtual procedure will generate code like:

```
if (x != null) // INSPECT x
      i = new Q(CUR, new RTS_NAME<RTS_PRCQNT>() {
            public RTS_PRCQNT get() { return (x.V()); }
      }).RESULT;
```

Where V() is the RTS_PRCQNT get() method used to evaluate the virtual match.

2.5. Some Expressions

2.5.1 Object Relation IS:

b := ***x is C***;

will be translated to:

```
b = IS(x,C.class);
```

where the method *IS* is defined by:

```
public boolean IS(final Object obj,final Class<?> cls) {
	return((obj == null)?false:(obj.getClass() == cls));
}
```

2.5.2 Object Relation IN:

b := ***x in C***;

will be translated to:

```
b = x instanceof C;
```

2.5.3 QualifiedObject QUA:

x qua B.p3 := 3.14;

will be translated to:

```
((B)x).p3=3.14;
```

2.6 Some Statements

2.6.1 Goto Statement

Java does not support labels like Simula. The Java Virtual Machine (JVM), however, has labels. A JVM-label is simply a relative byte-address within the byte-code of a method.

If the compiler generates byte code directly, the tableswitch and labelbinding instructions are generated directly into the ClassFile. No Byte Code Engineering is necessary in that case.

However; when creating Java source, we will use Java's exception handling together with *byte Code Engineering* to re-introduce goto in the Java Language.
This is done by generating Java-code which is prepared for Byte Code Engineering.

Suppose a Simula program containing labels and goto like this:

```
Begin
  L: ...
     goto L;
     ...
End;
```

This will be coded as:

```
1. public final class Example extends RTS_BASICIO {
2.
3.     final RTS_LABEL L = new RTS_LABEL(this, 0, 1, "L"); // Local Label #1=L
4.     public Example(RTS_RTObject staticLink) {
5.         super(staticLink);
6.         BBLK();
7.     } // End of Constructor
8.
9.     // SimulaProgram Statements
10.    public RTS_RTObject STM() {
11.        Example THIS = (Example)CUR;
12.        LOOP:while(JTX >= 0) {
13.            try {
14.                _JUMPTABLE(JTX, 1); // For ByteCode Engineering
15.                // Statements ....
16.                _SIM_LABEL(1,"L"); // L
17.                // Statements ....
18.                throw(L); // GOTO EVALUATED LABEL
19.                // Statements ....
20.                break LOOP;
21.            }
22.            catch(RTS_LABEL q) {
23.                RTS_RTObject._TREAT_GOTO_CATCH_BLOCK(THIS, q);
24.                JTX = q.index; continue LOOP; // GOTO Local L
25.            }
26.        }
27.        EBLK();
28.        return(null);
29.    }
30. }
```

The utility method *TREAT_GOTO_CATCH_BLOCK at line 23 will* test if the label q belongs to THIS RTObject and, if so, simply return, otherwise it will set the current object terminated and re-throw the label.

2.6.1.1 Label Quantities.

At source line 3. the label 'L' is declared like this:

```
final RTS_LABEL L = new RTS_LABEL(this, 0, 1, "L"); // Local Label #1=L
```

Where RTS_LABEL is defined by:

```
public final class RTS_LABEL extends RuntimeException {
    public RTS_RTObject SL; // Static link, block in which the label is defined.
    public final int _PRFX; // Prefix level.
    public int index; // Ordinal number of Label within its Scope(staticLink).
    public final String identifier; // To improve error and trace messages.

    public RTS_LABEL(RTS_RTObject SL, int index, final String identifier)  {
       this.SL=SL; this.index=index; this.identifier=identifier;
    }
}
```

A goto-statement is simply coded as:

```
throw(L); // GOTO EVALUATED LABEL
```

And this exception is caught and tested (lines 23 - 24) throughout the operating chain. If the label does not belong to this block instance the exception is rethrown.

In the event of no matching block instances the exception is caught by an UncaughtExceptionHandler like this:

```
public void uncaughtException(Thread thread, Throwable e) {
    if(e instanceof RTS_LABEL) {
        // POSSIBLE GOTO OUT OF COMPONENT
        RTS_RTObject DL=obj.DL;
        if(DL!=null && DL!=CTX) {
            DL.PENDING_EXCEPTION=(RuntimeException)e;
            DL.CORUT.run();
        } else {
            ERROR("Illegal GOTO "+e);
            ...
        } else ...
 }
```

Thus, when a QPS-component is left we raise the PENDING_EXCEPTION flag and resume next operating component. The resume-operations will retrow the exception within its Thread.

For further details see the source code of RTS_RTObject.java

2.6.1.2 Byte Code Engineering.

Byte Code Engineering is performed using the new ClassFile API.
See: https://openjdk.org/jeps/484

If the compiler generates byte code directly, the tableswitch and labelbinding instructions are generated directly into the ClassFile. No Byte Code Engineering is necessary in that case.

However; when creating Java source, Byte Code Engineering is necessary.

To easily modify the code, the Simula Compiler generates certain method call in the .java file:

```
- JUMPTABLE(JTX, n); // For ByteCode Engineering
```

This method-call is a placeholder for where to put in a Jump-Table.
The parameter 'n' is the number of cases.

JUMPTABLE will always occur before any any labels.

Try to locate the instruction sequence:

```
PREV-INSTRUCTION
GETFIELD JTX
ICONST n
INVOKESTATIC JUMPTABLE
NEXT-INSTRUCTION
```

And replace it by the instruction sequence:

```
PREV-INSTRUCTION
GETFIELD JTX
TABLESWITCH ... uses 'n' labels which is binded later.
NEXT-INSTRUCTION
```

```
- SIM_LABEL(n); // Label #n
```

This method-call is used to signal the occurrence of a Simula Label. The bytecode address is collected and some instruction are removed. The parameter 'n' is the label's ordinal number.

I.e. Try to locate the instruction sequence:

```
PREV-INSTRUCTION
ICONST n
INVOKESTATIC _SIM_LABEL
NEXT-INSTRUCTION
```

And replace it by the instruction sequence:

```
PREV-INSTRUCTION
LABELBINDING( case n )
NEXT-INSTRUCTION
```

This is the CodeTransform that performs the necessary changes to fix the labels and goto's:

```
final class SimulaCodeTransform implements CodeTransform {
  private List<SwitchCase> cases;
  private CodeElement prevElement;

  @Override
  public void atEnd(CodeBuilder builder) {
    if (prevElement != null)
      builder.with(prevElement);
  }

  @Override
  public void accept(CodeBuilder builder, CodeElement element) {
    if (element instanceof InvokeInstruction instr) {
      if (instr.name().equalsString("_JUMPTABLE")) {
          int tableSize = getConst(prevElement);
          prevElement = null;
          cases = new Vector<SwitchCase>();
          for (int i = 1; i <= tableSize; i++)
              cases.add(SwitchCase.of(i, builder.newLabel()));
          // Build the TableSwitch Instruction
          Label defaultTarget = builder.newLabel();
          builder
            .tableswitch(1, cases.size(), defaultTarget, cases)
            .labelBinding(defaultTarget);
          return;
      }
      if (instr.name().equalsString("_SIM_LABEL")) {
          int caseValue = getConst(prevElement);
          prevElement = null;
          LabelTarget target = (LabelTarget) cases.get(caseValue - 1).target();
          builder
            .labelBinding(target.label());
          return;
      }
    }
    if (prevElement != null) builder.with(prevElement);
    prevElement = element;
  }

  private int getConst(CodeElement element) {
    if (element instanceof ConstantInstruction instr) {
      // instr: ICONST n or BIPUSH or SIPUSH
      ConstantDesc val = instr.constantValue();
      return (((Integer) val).intValue());
    } throw new RuntimeException("..." + element);
  }
}
```

For more information, see Simula.compiler.transform

2.6.2 For-Statement

The Implementation of the for-statement is a bit tricky. The basic idea is to create a ForList iterator that iterates over a set of ForElt iterators. The following subclasses of ForElt are defined:

- SingleElt<T> for basic types T control variable
- SingleTValElt for Text type control variable
- StepUntil for numeric types
- WhileElt<T> for basic types T control variable
- WhileTValElt representing For t:= <TextExpr> while <Cond>
  With text value assignment

Each of which deliver a boolean value 'CB' used to indicate whether this for-element is exhausted. All parameters to these classes are transferred 'by name'. This is done to ensure that all expressions are evaluated in the right order. The assignment to the 'control variable' is done within the various for-elements when the 'next' method is invoked. To get a full overview of all the details you are encouraged to study the generated code together with the *'FRAMEWORK for for-list iteration'* found in the runtime class RTS_RTObject.

Example, the following for-statement:

```
for i:=1,6,13 step 6 until 66,i+1 while i<80 do j:=j+i;
```

Is compiled to:

```
for(boolean CB:new ForList(
    new SingleElt<Number>(...)
   ,new SingleElt<Number>(...)
   ,new StepUntil(...)
   ,new WhileElt<Number>(...)
)) { if(!CB) continue;
     j=j+i;
   }
```

Another example with control variable of type Text:

```
for t:="one",other while k < 7 do  <statement>
```

Where 'other' is a text procedure, is compiled to:

```
for(boolean CB:new ForList(
    new SingleTValElt(...)
   ,new WhileTValElt(...)
 )) { if(!CB) continue;
       … // Statement
    }
```

For further details see the source code of ForStatement.java.

Optimized For-Statement

However; most of the for-statements with only one for-list element are optimized.

Single for-step-until statements are optimized when the step-expression is constant. I.e. the following for-statements:

for i:=<expr-1> step 1 until <expr-2> do <statements>
for i:=<expr-1> step -1 until <expr-2> do <statements>

for i:=<expr-1> step 6 until <expr-2> do <statements>
for i:=<expr-1> step -6 until <expr-2> do <statements>

are compiled to:

```
for(i = <expr-1>; i <= <expr-2>; i++) { <statements> }
for(i = <expr-1>; i >= <expr-2>; i--) { <statements> }

for(i = <expr-1>; i <= <expr-2>; i=i+6) { <statements> }
for(i = <expr-1>; i >= <expr-2>; i=i-6) { <statements> }
```

The other kinds of single elements are optimized in these ways:

for i:=<expr> do <statements>

for i:=<expr> while <cond> do <statements>

are compiled to:

```
i = <expr>; { <statements> }

i = <expr>;
While( <cond> ) {
       <statements>;
       i = <expr>;
}
```

2.6.3 Connection Statement

The connection statement is implemented using Java's **instanceof** operator and the **if** statement. For example, the connection statement:

inspect x **do** image:-t;

Where 'x' is declared as a reference to an ImageFile, is compiled to:

```
if(x!=null) x.image=t;
```

Other examples that also use 'ref(Imagefile) x' may be:

1) **inspect** x **do** image:-t **otherwise** t:-**notext**;

2) **inspect** x
   when infile **do** t:-intext(12)
   when outfile **do** outtext(t);

3) **inspect** x
   when infile **do** t:-intext(12)
   when outfile **do** outtext(t)
   otherwise t:-**notext**;

These examples are compiled to:

1)
```
if(x!=null) x.image=t; else t=null;
```

2)
```
if(x instanceof RTS_Infile) t=((RTS_Infile)x).intext(12);
else if(x instanceof RTS_Outfile) ((RTS_Outfile)x).outtext(t);
```

3)
```
if(x instanceof RTS_Infile) t=((RTS_Infile)x).intext(12);
else if(x instanceof RTS_Outfile) ((RTS_Outfile)x).outtext(t);
else t=null;
```

For further details see the source code of ConnectionStatement.java.

2.6.4 Switch Statement

The Switch Statement is a language extension inherited from S-Port Simula.

Syntax:

Switch-statement
= **SWITCH** (lowKey : hiKey) switchKey **BEGIN** { switch-case } [none-case] **END**

switch-case = **WHEN** caseKey-list **do** statement ;
none-case = **WHEN NONE do** statement ;
caseKey-list = caseKey { , caseKey }
caseKey = caseConstant | caseConstant : caseConstant
lowKey = integer-or-character-expression
hiKey = integer-or-character-expression
switchKey = integer-or-character-expression
caseConstant = integer-or-character-constant

Translated into a Java Switch Statement with break after each <statement>.

For example, the following Switch Statement::

```
switch(lowkey:hikey) key
begin when 1 do !Statement;
      when 2 do !Statement;
      when NONE do !Statement;
      end;
end switch;
```

Is compiled to:

```
if(key<lowkey || key>hikey) throw
      new RTS_SimulaRuntimeError("Switch key outside key interval");
switch(key) {
      case 1:
            // Statement
            break;
      case 2:
            // Statement
            break;
      default:
            // Statement
            break;
}
```

For further details see the source code of SwitchStatement.java.

2.6.5 Activation Statement

The activation statement is defined by the procedure ACTIVAT in Simula Standard. In this implementation we use a set of methods for the same purpose:

activate x;	==>	`ActivateDirect(false,x);`
activate x delay 1.34;	==>	`ActivateDelay(false,x,1.34f,false);`
activate x delay 1.34 prior;	==>	`ActivateDelay(false,x,1.34f,true);`
activate x at 13.7;	==>	`ActivateAt(false,x,13.7f,false);`
activate x at 13.7 prior;	==>	`ActivateAt(false,x,13.7f,true);`
activate x before y;	==>	`ActivateBefore(false,x,y);`
activate x after y;	==>	`ActivateAfter(false,x,y);`
reactivate x;	==>	`ActivateDirect(true,x);`
reactivate x delay 1.34;	==>	`ActivateDelay(true,x,1.34f,false);`
reactivate x delay 1.34 prior;	==>	`ActivateDelay(true,x,1.34f,true);`
reactivate x at 13.7;	==>	`ActivateAt(true,x,13.7f,false);`
reactivate x at 13.7 prior;	==>	`ActivateAt(true,x,13.7f,true);`
reactivate x before y;	==>	`ActivateBefore(true,x,y);`
reactivate x after y;	==>	`ActivateAfter(true,x,y);`

See runtime module RTS_Simulation for details.

For further details see the source code of ActivationStatement.java.

2.7. Sequencing

2.7.1 Co-routines: detach - call

Simula has three primitives to express Sequencing:

1) obj.Detach – Suspends the execution, saves a reactivation point and returns.
2) Call(obj) – Restarts a detached object at the saved reactivation point.
3) Resume(obj) – Suspend and restart another object.

The (detach,call) pair constitutes co-routines in Simula while the (detach,resume) pair establish Symmetric component sequencing used to implement discrete event simulation. For details on the implementation, see the runtime system code.

A small Java class 'Coroutine' is used to implement the qps primitives.

```
public class Coroutine implements Runnable {
    public Coroutine(Runnable target)
    public static Coroutine getCurrentCoroutine()
    public final void run()
    public static void detach()
    public boolean isDone()
    ....
}
```

A coroutine is a program object representing a computation that may be suspended and resumed. The implementation use Virtual Threads to constitute separate stacks.

To illustrate the transition of the pointers when executing the qps primitives we will use the following program example.

```
System begin ref(C) x;
        procedure P;
        begin x:-new C();
                …
                call(x);
        end;
        class Component;
        begin
                procedure Q;
                begin ... detach; ... end;
                ...
                Q;
                ...
        end Component;
        ...
        P;
        ...
end System;
```

The following pages show the transitions in detail.

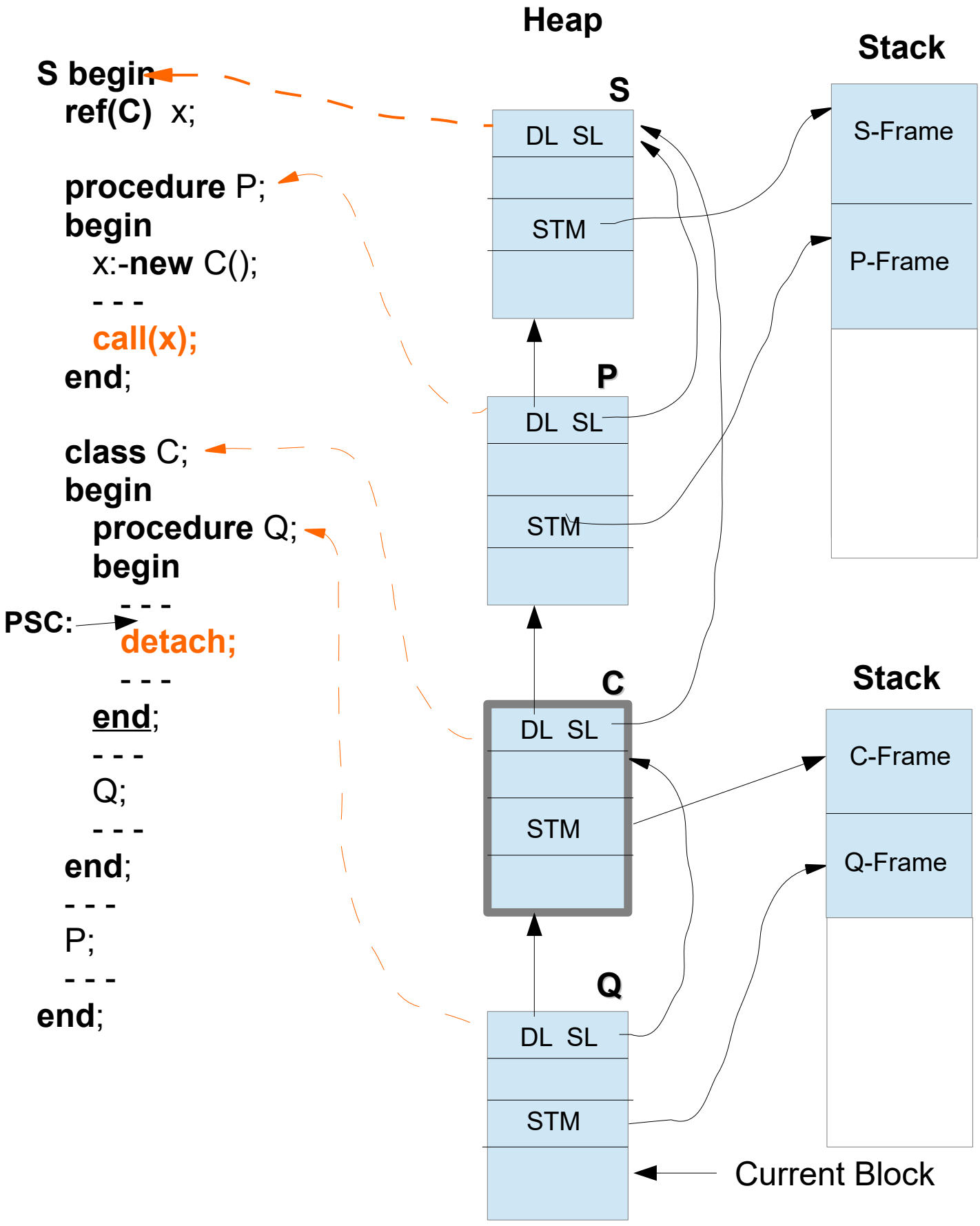


Fig. 2: Just before Detach

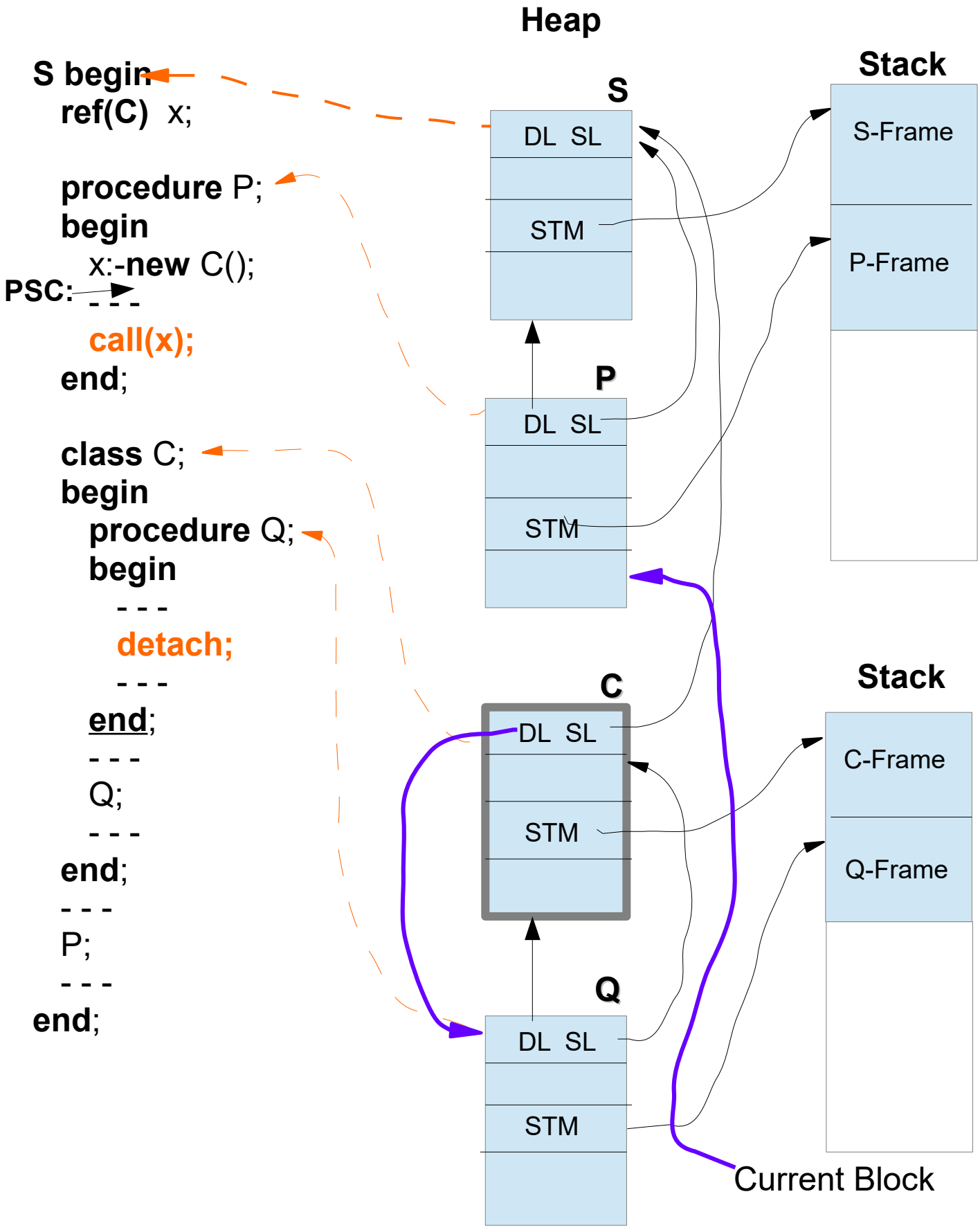


Fig. 3: Just after Detach – C-Object is dismounted

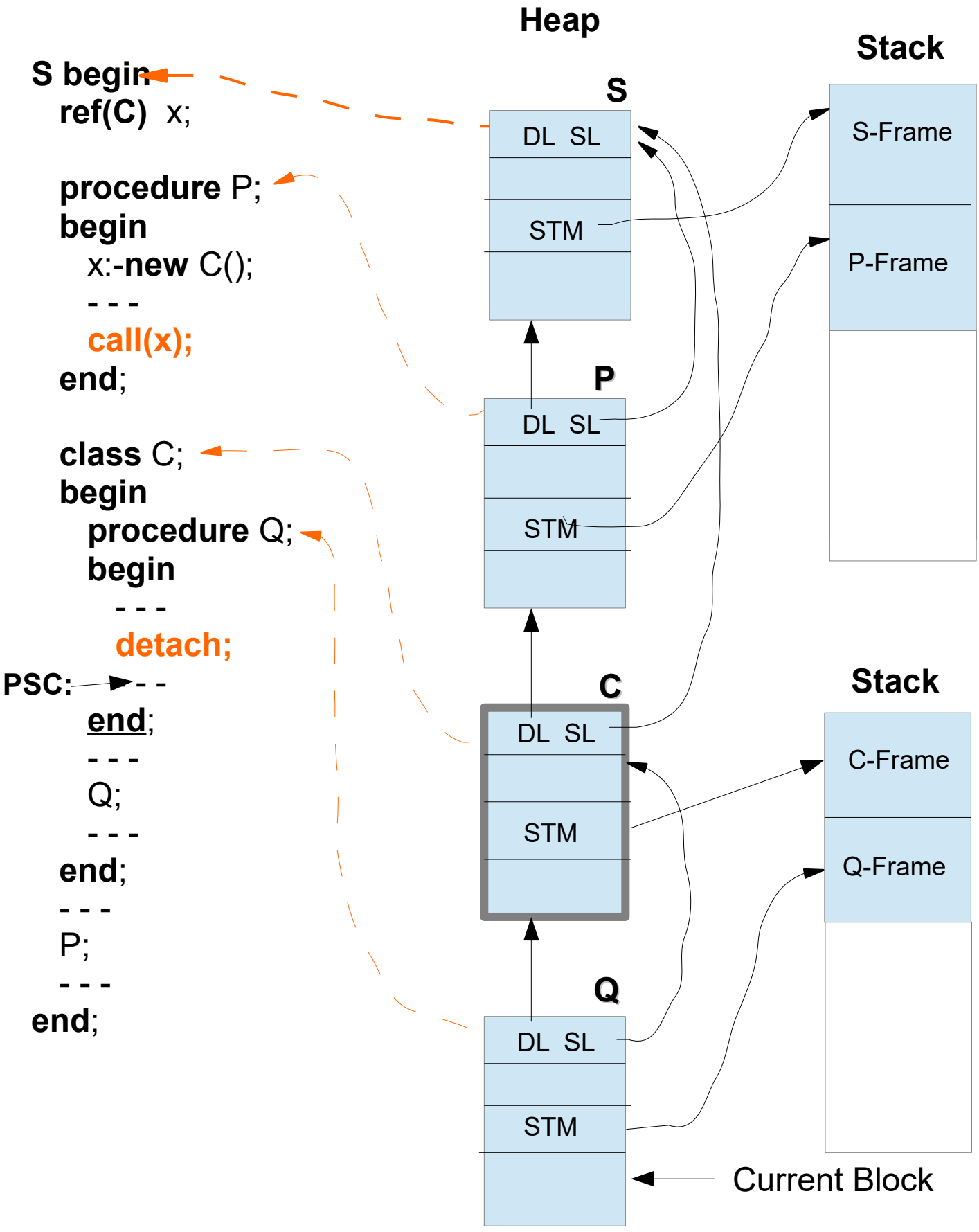


Fig. 4: Just after Call – C-Object is re-mounted

2.7.2 Symmetric component sequencing: detach - resume

Symmetric component sequencing, the (detach,resume) pair, is used to implement discrete event simulation. This is a complicated implementation that was largely inherited from Simula I. It is believed that this should have been implemented using ordinary coroutines.

For details on the implementation, see the runtime system code.

To illustrate the transition of the stack when executing the qps primitives we will use the following program example.

```
System begin ref(C) x;
        procedure P;
        begin x:-new C();
                …
                resume(x);
        end;
        class Component;
        begin
                procedure Q;
                begin
                        while … do
                        begin ... detach; … end;
                end;
                ...
                Q;
                ...
        end Component;
        ...
        P;
        ...
end System;
```

The following pages show the transitions in detail.

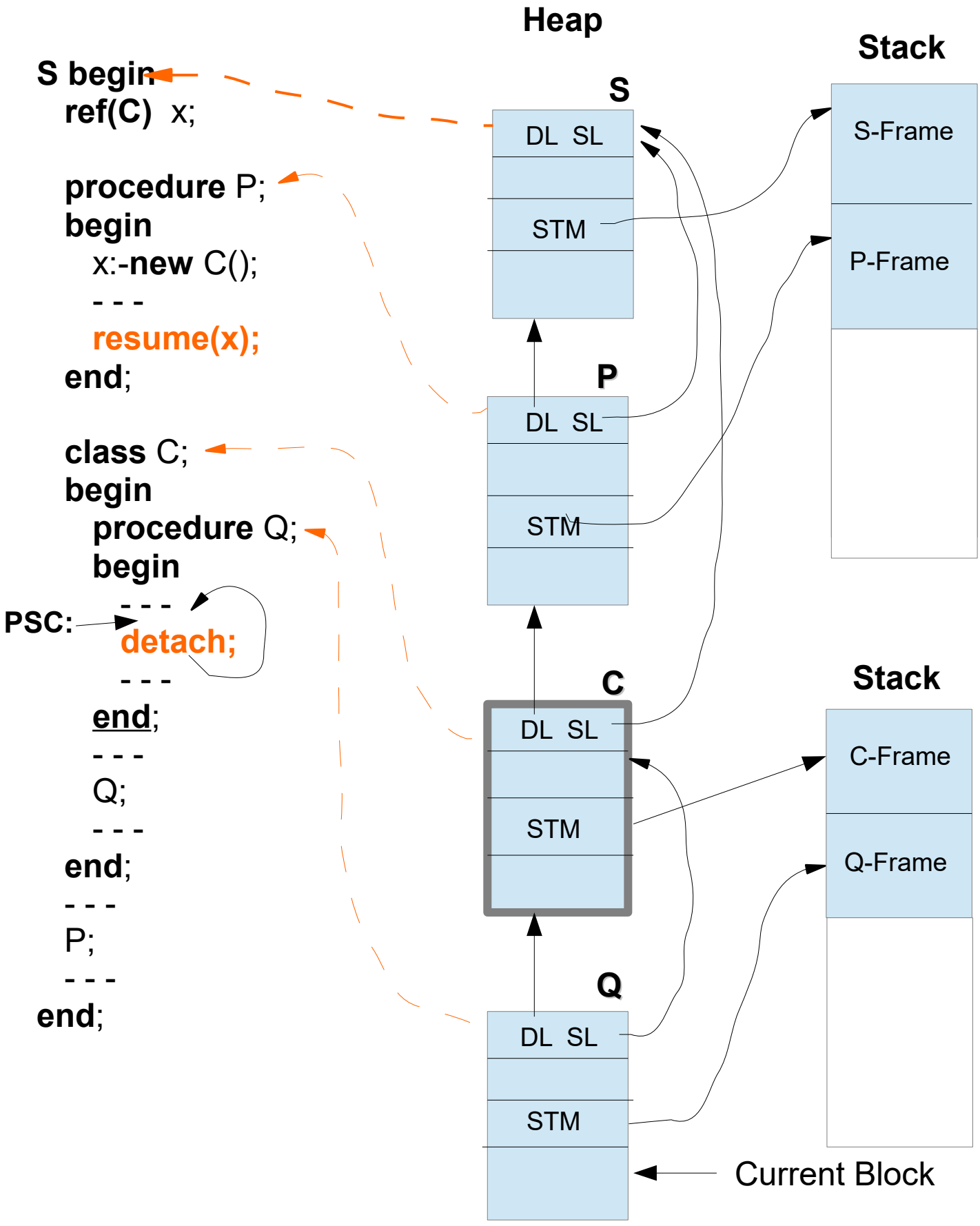


Fig. 5: Just before 1. Detach

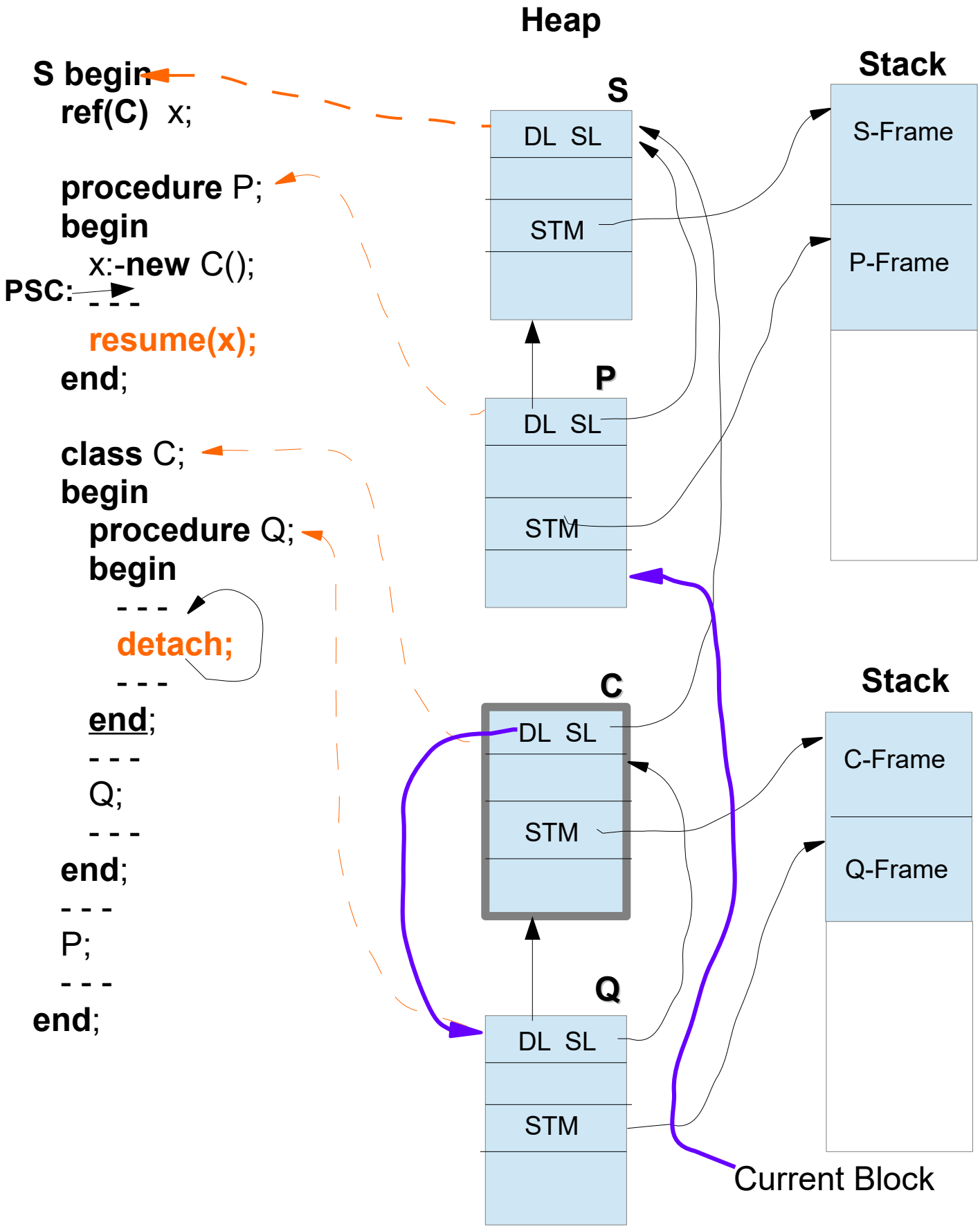


Fig. 6 Just after 1.Detach and before Resume

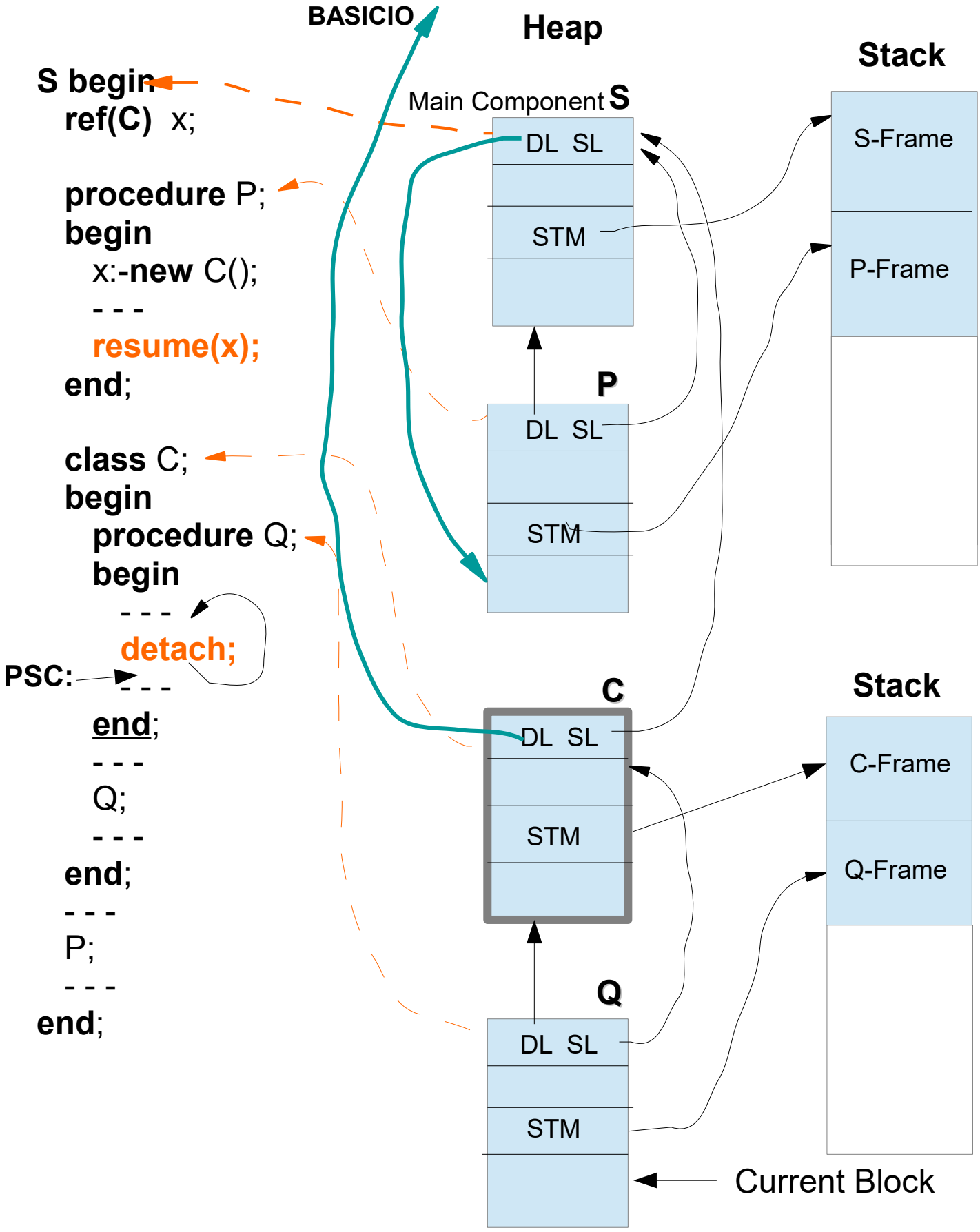


Fig. 7: Just after Resume

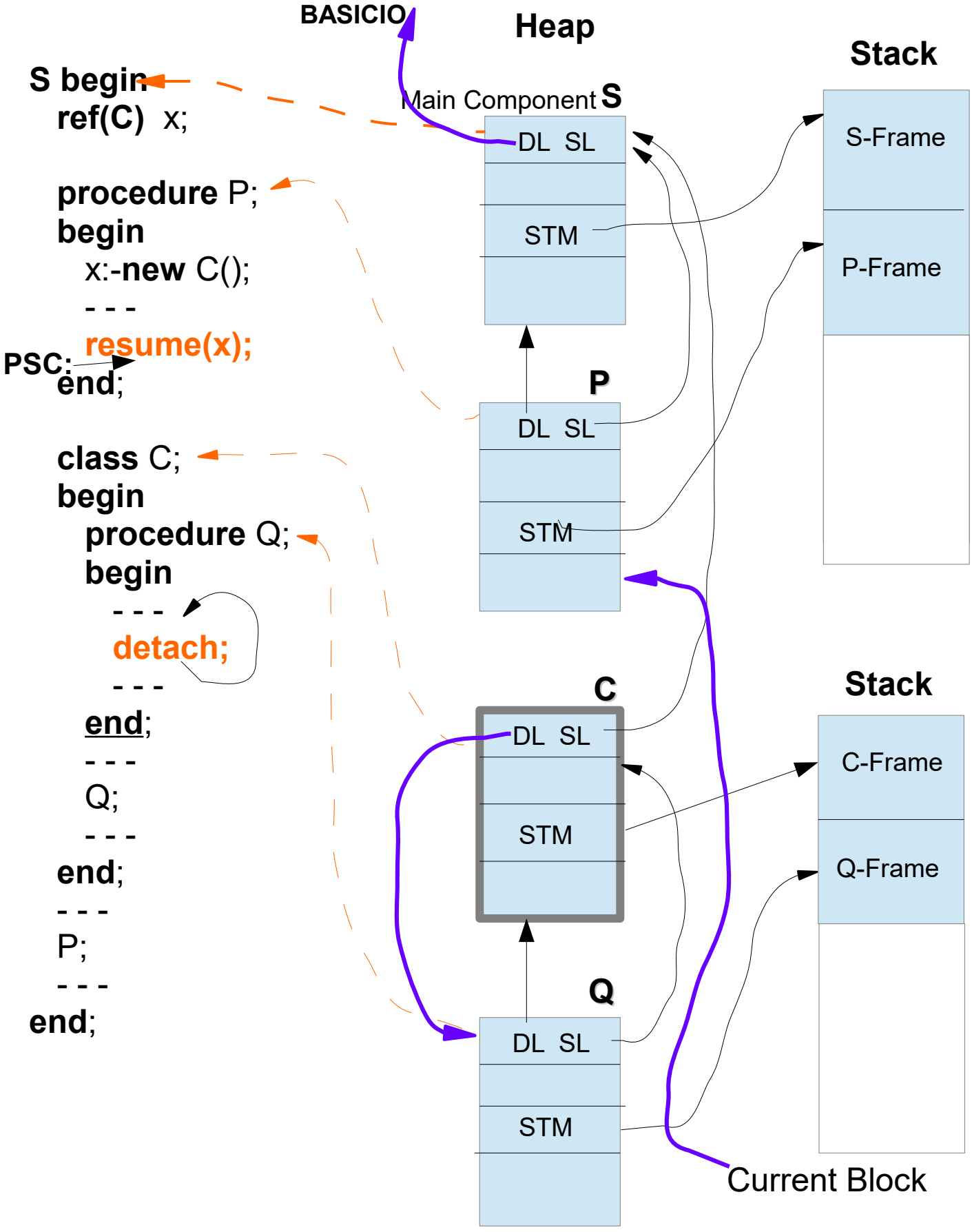


Fig. 8: Just after 2. Detach

2.7.3 Definition of Detach, Call and Resume

In this implementation, detach, call and resume are defined in the class RTS_RTObject. The method used is due to the work of Arne Wang in 1982 where he introduced two primitives *swap* and *rotate* to descibe these methods.

```
public void detach() {
	if (isQPSystemBlock())
		return; // Detach QPS System Block is no-operation.
	RTObject CUR = null;
	// Make sure that this object is on the operating chain.
	// Note: A detached or terminated object cannot be on the operating chain.
	RTObject dl = CUR;
	while (dl != this) {
		dl = dl.DL;
		if (dl == null)
			error("x.Detach: x is not on the operating chain.");
	}
	switch (this.STATE) {
	case resumed -> {
		// Find main component for component to be detached. The main
		// component head must be the static enclosure of the object.
		RTObject main = this.SL;
		// Rotate the contents of 'CUR', 'this.DL' and 'main.DL'.
		// <main.DL,this.DL,CUR> := <this.DL,CUR,main.DL>
		dl = main.DL;
		main.DL = this.DL;
		this.DL = CUR;
		CUR = dl;
	}
	case attached -> {
		// Swap the contents of object's 'this.DL' and 'CUR'.
		// <this.DL,CUR> := <CUR,this.DL>
		dl = this.DL;
		this.DL = CUR;
		CUR = dl;
	}
	default -> error("Illegal Detach");
	}
	this.STATE = OperationalState.detached;

	// ... other stuff
}
```

For further details see the source code of Detach.

```
public void call(final RTObject ins) {
	RTObject dl; // Temporary reference to dynamic enclosure.
	if (ins == null) error("Call(x): x is none.");
	if (ins.STATE != OperationalState.detached)
		error("Call(x): x is not in detached state.");
	// The object to be attached cannot be on the operating chain,
	// because then its state would have been resumed and not detached.

	// Swap the contents of 'CUR' and object's 'dl'.
	// <ins.DL,CUR>:=<CUR,ins.DL>;
	dl = ins.DL;
	ins.DL = CUR;
	CUR = dl;
	// From now on the object is in attached state.
	// It is no longer a component head.
	ins.STATE = OperationalState.attached;

	// ... other stuff
}
```

For further details see the source code of Call.

```
public void resume(final RTObject ins) {
	RTObject comp;   // Component head.
	RTObject mainSL; // Static enclosure of main component head.
	RTObject main;   // The head of the main component and also
	                 // the head of the quasi-parallel system.
	if (ins == null) error("Resume(x): x is none.");
	if (ins.STATE != OperationalState.resumed) { // A no-operation?
		// The object to be resumed must be local to a system head.
		main = ins.SL;
		if (!main.isQPSystemBlock())
			error("ins is not local to sub-block or prefixed block.");
		if (ins.STATE != OperationalState.detached)
			error("ins is not in detached state but");
		// Find the operating component of the quasi-parallel system.
		comp = CUR;
		mainSL = main.SL;
		while (comp.DL != mainSL)
			comp = comp.DL;
		if (comp.STATE == OperationalState.resumed)
			comp.STATE = OperationalState.detached;
		// Rotate the contents of 'ins.dl', 'comp.dl' and 'CUR'.
		// Invariant: comp.DL = mainSL
		// <ins.DL,comp.DL,CUR>=<comp.DL,CUR,ins.DL>
		comp.DL = CUR;
		CUR = ins.DL;
		ins.DL = mainSL;
		ins.STATE = OperationalState.resumed;

		// ... other stuff
	}
}
```

For further details see the source code of Resume.