Language Interoperability with BETA on .NET and Java Virtual Machines

Ole Lehrmann Madsen & Peter Andersen
Computer Science Department, Aarhus University
Åbogade 34, DK-8200 Århus N, Denmark

June 18, 2004

Abstract

Note! Current version of how BETA is mapped into CLR/JVM

1 Introduction

The overall goal of the project presented in this paper is language interoperability between object-oriented languages. From a language design/theoretical point-of-view issues concerning language interoperability are interesting since they highlight essential similarities and differences between languages. From an implementation point of view, design and implementation of execution platforms (virtual machine, common run-times) supporting a wide range of languages are interesting. From a practical point-of-view it is interesting to be able to reuse libraries and frameworks between across language borders.

Note! Interesting for the following reasons ...

The work reported here describes an exercise in implementing the BETA language on Microsoft's .NET platform and the Java virtual machine platform. BETA is an object-oriented programming language and implementations in the form off native compilers exists for a number of platforms just as a powerful IDE – the Mjølner System – is available, see www.mjolner.dk. For BETA all three issues mentioned above are interesting.

At the language design level it is interesting to explore to what extent languages are similar and just appear different due to syntactic issues. It is interesting to explore to what extent mechanism found in one language can be simulated or abstracted in another language. And finally it is interesting to find out to what extent mechanisms differ fundamentally between languages. For BETA the the project has contributed to achieve a better understanding of the parts of BETA that differ from other OO languages and it has contributed with knowledge about which primitives and abstraction mechanisms should be part of a language covering BETA and say the Java and C# family of languages.

Virtual machines with just-in-time compilers are becoming more and more common as the basic technology for implementing object-oriented languages. For some years Java from Sun Micro systems has been the dominant technology in industry, but with Microsoft's .NET platform, an alternative has arrived. Both platforms are based on virtual machines with a bytecode instruction set and type information that makes it possible to verify the safety of a given program. The Java platform has been designed solely for supporting the Java language although there exist implementations for other languages. The .NET-platform on the other hand was designed to support language interoperability. On .NET it is e.g. possible to use a class written in one language and make a subclass of it in another language. The C# language has been designed for the .NET-platform and is in many ways similar to the Java language.

The porting of BETA to .NET and JVM has the following goals:

- To get experience with implementing a language like BETA on these platforms. BETA differs in many respects from Java and C# and since the Java- and .NET platforms are both designed for implementing Java- and C#-like languages, there may be parts of BETA that can not be easily implemented.

- To be able to evaluate and compare the two platforms. It is interesting to investigate whether there are significant differences between the two platforms.

- To test language interoperability on these platforms. One goal is to find out if language interoperability does work as promised by .NET. Another goal is to find out if language interoperability works with BETA and the other .NET languages. Since BETA differs significantly from e.g. C# it is not obvious whether it is useful in practice to use a BETA pattern (class) in C# or Visual Basic. Another goal is to experiment with language interoperability on the Java platform.

- To offer a language BETA that can be used to write applications for both platforms. If the porting of BETA to both platforms succeed, it will be possible to use BETA for implementing applications that can run on both platforms.

In [1] work on integrating BETA with the Eclipse integrated Development Environment (IDE) and Visual Studio has been reported. This is another example of language interoperability.

2 An immediate mapping of BETA to CLR/JVM

There have been two issues regarding the mapping of BETA to CLR/JVM:

1. CLR/JVM are typed virtual machines designed to support Java for JVM and a number of languages like C#, and VB for CLR. BETA is much more general than Java and C# so one issue has been to find a mapping that works.

2. The other issue is language interoperabiliby as seen from the point-of-view of programmers. To obtain true language interoperability, is should be possible to use classes written in Java, C#, etc. from BETA and vice versa. Given an implementation of BETA for CLR/JVM this should be possible. The main issue will be readability/understandability of doing this. Using Java and C# from BETA will probably not pose problems, since a BETA programmer just have to understand the Java/C# classes being used.

   The other way around may be more problematic. Since BETA is much more general than Java and C#, the mapping of BETA patterns to CLR/JVM may without careful design be complicated and result in class-files that are difficult to understand at the level of Java and C#.

The mapping of BETA into CLR/JVM is described as as a mapping into a common subset of C# and Java. The subsequent mapping into CLR/JVM bytecodes should then appear as straight forward. That is a the C#/Java level the mappings to the two platforms are essentially identical. There are of course differences when considering the bytecode level but these are purely technical – the logical structure is identical.

The language model – as we from now on call *CJ* – behind Java and C# may be characterized in the following way:

- A program is a collection of classes where each class defines the structure of a set of objects.
- A class defines
  - A set of data-items – instance variables belonging to instances of the class, and static variables belonging to the class.
  - A set of methods – instance methods and static methods.
- Classes may be textually nested: in Java real nesting in the form of block-structure as in Simula and BETA is supported whereas C# only provides nesting as a scoping mechanism similar to C++.
- Methods may be overloaded.
- There is support for dynamic exceptions.
- Concurrency is supported in the form of threads and a monitor-like construct.

The BETA language may be characterized in the following way:

- Class and method have been unified into the pattern mechanism.
- Patterns can be arbitrarily nested supporting general block-structure.
- The INNER-mechanism is used to combine methods instead of super.
- Genericity is supported by means of virtual patterns.
- A pattern (method) invocation may return multiple values.
- Coroutines and concurrency is based on semi-coroutines as known from Simula in the form of active objects. Basic synchronization between concurrent objects is supported by a semaphore pattern. These mechanisms are at a more basic level than threads and monitors in CJ, but form a basis for implementing higher-level concurrency abstractions like monitors and Ada-like rendezvous and schedulers.
- BETA has no constructor mechanism.
- Methods cannot be overloaded.
- BETA is based on a static exception handling mechanism [5] and dynamic exceptions are only supported in an experimental version of the language.

Note that this paper is not an attempt to present the rationale behind the generality of BETA compared to main stream languages like C# and Java. The purpose is to present the issues in mapping BETA to CLR/JVM. To do this we will use more or less useful examples to illustrate the mapping. The rationale behind BETA has been presented in a number of articles such as [7, 6].

Consider the following example of a BETA pattern describing a simple bank account:

```
Account:
  (# balance: @integer;
     deposit:
       (# amount: @integer
       enter amount
       do balance + amount -> balance
       exit balance
       #);
     withdraw:
       (# amount: @integer enter amount do balance - amount -> balance #);
  #);
```

The `Account` pattern has three attributes `balance`, `deposit` and `withdraw`. The attribute `balance` is an instance variable holding the current value of the `Account`. The attributes `deposit` and `withdraw` describe operations on the account. The operation `deposit` inserts an amount on the account by adding to `balance` to the value of the enter-parameter `amount` and returns (exit) the value of `balance`. The operation `withdraw` similarly withdraws an amount from the account.

The following example shows an instance `myAccount` of `Account`, an instance `X` of `integer`; a call of `myAccount.deposit` with enter arguments `120` and the resulting exit-value being assigned to `X`; and a call of `myAccount.withdraw` with enter argument `50`.

```
myAccount: @Account; X: @integer;
120 -> myAccount.deposit -> X;
50 -> C.withdraw;
```

Note that `Account`, `deposit` and `withdraw` are all examples of patterns. In this example `Account` is used as a class and `deposit` and `withdraw` as methods. This use of the `Account` pattern is illustrated by the following simple mapping to a CJ class:

```
class Account extends Object {
   int balance;
   int deposit(int amount) {
      balance = balance + amount;
      return balance;
   }
   int withdraw(int amount) { balance = balance - amount; }
}
```

The BETA declarations and invocations shown above then maps to the following declarations and invocations in CJ:

```
Account myAccount = new Account(); int X;
X = myAccount.deposit(120);
myAcount.withdraw(50);
```

The above mapping shows a simple semantics of the BETA `Account` pattern. To capture the full semantics of the `Account` pattern a more complex mapping is needed. The fact that e.g. `deposit` is a pattern means that it is possible to use `deposit` as a class and create instances of `deposit` as in the following example:

```
myAccount: @Account;
myDeposit: ^myAccount.deposit;
myAccount.deposit[] -> myDeposit[];
```

The variable `myDeposit` may refer to instances of `myAccount.deposit`. The statement `&myAccount.deposit[] -> myDeposit[]` creates an instance of the pattern `myAccount.deposit` and assigns its reference to `myDeposit`.

Using `myDeposit`, the instance variable `amount` in `myDeposit` may be assigned a value as in:

```
170 -> myDeposit.balance
```

The pattern `deposit` also defines a do-part, which is executed when `deposit` is used a a procedure. The do-part of `myDeposit` may be executed directly by the following statement (the exact semantics should be clear when we show the complete mapping of the `Account` pattern below):

```
20 -> myDeposit -> X
```

To obtain the full semantics of the `deposit` pattern, it is mapped into the following inner class[1] of `Account`:

```
class deposit extends Object {
   int amount;
   void enter(int a) { amount = a; }
   void do() { balance = balance + a; }
   int exit() { return balance; }
}
```

A BETA invocation

```
 50 -> myAccount.deposit -> X
```

is then implemented as follows :

```
deposit D = new deposit();
D.enter(50);
D.do();
X = D.exit();
```

Note! Just show the method below

To avoid generating the above code for each invocation, and `deposit`-method is generated as part of class `Account`:

```
int deposit(int amount) {
    deposit D = new deposit();
    D.enter(amount);
    D.do();
    return D.exit();
}
```

The BETA invocation `50 -> myAccount.deposit -> X` may now be mapped into the corresponding CJ invocation `X = myAccount.deposit(50)`.

Note! Drop new-method here. Check def/use of call- and new-method

As mentioned,it is also possible to generate instances of the patter `deposit`. To handle this a *new*-method of the following form is generated as part of class `Account`.

[1]The reader may observe that this use of an inner class will work in Java, but not in C#. This issue is further discussed below.

```
deposit deposit() { return deposit(); }
```

Instances of `deposit` may now be generated by execution of `myDeposit = myAccount.deposit()`.

The complete mapping of the `Account` pattern is shown below:

```
class Account extends Object {
  int balance = 0;
  int deposit(int amount) {
    deposit D = new deposit();
    D.enter(amount);
    D.do();
    return D.exit();
  }
  void withdraw(int amount) { ... };
  class deposit extends Object { ... };
  class withdraw extends Object { ... };
}
```

As can be seen, each inner pattern of `Account` – in this case `deposit` and `withdraw` – gives rise to a method and an inner class.

The method is for using the pattern as a method – for `withdraw` this is the method with the signature `int withdraw(int amount)`.

The `Account` example also shows how to use BETA patterns from Java and C#. A Java or C# class may use the `Account` class as shown above and invoke the `deposit`- and `deposit`-methods as shown in the following Java example:

```
Account myAccount = new Account();
deposit myDeposit = myAccount.new deposit()
mydeposit.enter(170);
mydeposit.do();
int X = myDeposit.exit();
```

In the following sections we will more systematically show how BETA constructs are mapped into CJ.

3 The general scheme for mapping a pattern

Note! Consider skipping this part – is very sketchy

Consider a pattern with nested patterns:

```
Foo:
   (# ...
      f1: (# ... enter N1 do ... exit X1 #);
      f2: (# ... enter N2 do ... exit X2 #);
      f3: (# ... enter N3 do ... exit X3 #);
   enter N
   do ...
   exit X
   #)
```

Note! We still need to introduce the new-method

Pattern Foo is mapped into the following class:

```
class Foo extends Object {
   ...
   sX1 f1(sN1) { ... } // call-method for f1
   f1 new$f1() { ... } // new-method for f1

   sX2 f2(sN2) { ... } // call-method for f2
   f2 new$f2() { ... } // new-method for f2

   sX3 f3(sN3) { ... } // call-method for f3
   f3 new$f3() { ... } // new-method for f2

   void enter(sN) { ... }
   void do() { ... }
   sX exit() { ... }

   class f1 extends Object { ... }
   class f2 extends Object { ... }
   class f3 extends Object { ... }
}
```

where sX1, sN1, ..., sN, and sX are the signatures corresponding to X1,N2, ... N, and X respectively.

4 Mapping nested patterns

Consider the following structure of nested (inner) patterns:

```
A: (# ...
      AA: (# ...
             AAA: (# ... #)
          #)
   #)
```

This structure is mapped into the following structure of nested CJ-classes:

```
class A extends Object {
   class AA extends Object {
      ...
      class AAA extends Object {
         ...
      }
   }
}
```

The above scheme is semantically correct for Java, but does not work for C#. In C# instances of inner classes cannot refer to instance variables in the enclosing object. To handle this BETA and Java maintains a structural reference to the enclosing object of an inner class. For C#, we have to generate this reference as part of the mapping. The above nested BETA patterns are thus mapped into the following C# structure:

```
class A extends Object {
   class AA extends Object {
      A origin;
      AA(A org) { origin = org;}
      ...
      class AAA extends Object {
         AA origin;
         AAA(AA org) { origin = org; }
         ...
      }
   }
}
```

Each inner class has an explicit reference, `origin` to its enclosing object. The origin reference is setup by a constructor that has the origin as an argument.

Note! Figure with nested A, AA, and AAA instances?

The `Account` example thus maps into the following classes for C#:

```
   class Account extends Object {
      int balance = 0;
      int deposit(int amount) {
          deposit D = new deposit(this);
          D.enter(amount);
          D.do();
          return D.exit();
      }
      void withdraw(int amount) { ... };
      class deposit extends Object {
         origin Account;
         int amount;
         deposit(origin org) { origin = org; }
         void enter(int a) { amount = a; }
         void do() { origin.balance = origin.balance + amount; }
         int exit() { return origin.balance; }
      }
      class withdraw extends Object { ... };
}
```

Note! Some explanation ...

An inspection of the class-files and generated by Java (and C#?) reveals that inner classes are flattened as outer classes and thus all appear at the same level of block structure. To handle this a name mangling scheme is used by Java (and C#?) to ensure uniqueness of class names. The above BETA patterns then at the class-file level are mapped into the following flat structure of classes:

```
class A extends Object { ... }
class A$AA extends Object {
   A origin;
   A$AA(A org) { origin = org; }
   ...
}
class A$AA$AAA extends Object {
   A$AA orgin;
   A$AA$AAA(A$AA org) { origin org; }
   ...
}
```

The flattening, name mangling and explicit origin reference are only a matter at the class-file level. At the user level – Java and C# programmers using BETA, the flattening does not show up. Except that C# programmers have to deal with the origin reference.

> *Note! We should either make use of Java inner classes or have a remark about it. I.e. that we currently do not use them but plan to.*

5 Mapping subpatterns and inner

In BETA a pattern can be a subpattern of another pattern just as a class can be a subclass of another class. A subpattern can be mapped directly to a subclass in CJ.

The inner-mechanism of BETA used for combination of the do-parts of a pattern and its superpattern cannot be directly mapped into CJ. In the following example, an inner has been added to the `withdraw` pattern of `Account`. In addition an `owner` attribute and a subpattern `depositWithNotify` of `withdraw` has been added.

> *Note! A better example is perhaps needed: with code before and after inner*

```
Account:
   (# balance: @integer;
      owner: ^Person;
      deposit:
       (# amount: @integer
       enter amount
       do balance + amount -> balance;
          inner deposit
       #);
```

```
        depositWithNotify: withdraw
          (#
          do amount -> owner.notify
          #)
     #)
```

Execution of `60 -> myAccount.depositWithNotify` implies that the do-part of `deposit` is executed followed by an execution of the do-part of `depositWithNotify`. This is captured by the following mapping into CJ:

```
class deposit extends Object {
   ...
   void do() {
      balance = balance + amount;
      do_1(); // inner deposit
   }
   protected void do_1()
}
class depositWithNotify extends deposit {
   void do_1() {
      owner.notify(amount);
   }
}
```

In general there may be an arbitrary number of subpatterns and inners and there may be statements to execute before and after an inner. This is illustrated in the following example:

Note! Alternative example to A,AA,AAA

```
Account:
  (#
     Open: Request
       (# name: ^text
       enter name[]
      do name[] -> R.open;
         inner;
         R.close
       #);
     ...
  #);
 (someR[],'foo',T[]) -> Use(# do T[] -> R.puttext #)
```

Note! Old example

```
 A: (# do X1; inner; Y1 #);
 AA: A(# do X2; inner; Y2 #);
 AAA: AA(# do X3; inner; Y3 #);
```

which will be mapped into:

```
class A extends Object {
   void do() { X1(); do_1(); Y1(); };
   protected void do_1();
}
class AA extends A {
   protected void do_1() { X2(); do_2(); Y2(); };
   protected void do_2();
}
class AAA extends AA {
   protected void do_2() { X3(); do_3(); Y3(); };
   protected void do_3();
}
```

Consider instances:

```
A a = new A(); AA aa = new AA(); AAA aaa = new AAA();
```

Note! text is missing for the next example

Execution of the do-method of theses objects takes places as follows

```
a.do();   A.do(); X1(); ...
```

As can be seen, inner is the empty action if there are no subpatterns of a given pattern.

Note! Example of a call should be shown somewhere

6 Mapping virtual patterns

A BETA pattern may be virtual. A virtual pattern used as as procedure corresponds to a (virtual) method in Java and C#. A virtual pattern used as a class corresponds to a class parameter of a generic class (parameterized class) in Java – there is no counterpart in C# (Is this correct?).

Note! More references to virtual class literature

The following example shows the use of virtual patterns. The pattern `Buffer` is a generic buffer that may hold an instance of the pattern `elm`. Pattern `elm` is declared as a virtual pattern attribute of `Buffer` – the `<` in `elm:< object` indicates that `elm` is a virtual attribute – the `object` part of the declaration indicates that `elm` may be extended in subpatterns of `Buffer` to subpatterns of `object`. This means that the type (qualification) of `elm` in `Buffer` is `object` and that `elm` can be used as if it is pattern `object`.

The instance variable `theElm`, the enter-argument `E` of `insert` and the exit-value `X` of `get` are all declared to be of type `elm`, i.e. in `Buffer`, `object`.

Since `object` is the most general superpattern in BETA, only few properties of `elm` is known in `Buffer`, but in subpatterns of `Buffer`, `elm` may be extended to any other pattern.

Note! display - currently ignored. Terminology: global versus local and nested patterns

```
Buffer:
  (# elm:< object;
     theElm: ^elm;
     put:
       (# E: ^elm
       enter E[] do E[] -> theElm[]
       #);
     get:
       (# X: ^elm
       do theElm[] -> X[]
       exit E[]
       #);
     display:<
       (#
       do 'Buffer: ' -> screen.puttext; inner;
       #)
  #);
```

Since `elm` is qualified by `object` arbitrary objects may be inserted into a `Buffer` as shown in the following example where an `Account` and a `Person` are inserted into a `Buffer`-object.

```
(# B: @Buffer;
   Joe: @Person;
   myAccount: @Account
   X: ^object;
do myAccount[] -> B.put;
   Joe[] -> B.put;
   B.get -> X[];
#)
```

The statement `B.get -> X[]` retrieves the element from the buffer and assigns it to `X`. Since `get` returns a reference of type `elm` the type of `X` also has to be `object`

Also the type of en element retrieved from the buffer by `get` is only know to be of type `object`.

In pattern `AccountBuffer` – a subpattern of `Buffer` – `elm` is extended to be the pattern `Account` as specified by the declaration `elm::< Account`. This means that inside `AccountBuffer`, we know that the instance variable `theElm` is at least of type `Account`. This is also the case for the enter-argument `E` of put and the exit-value `X` of get.

Note! We know more about elm

```
AccountBuffer: Buffer
  (# elm::< Account
     display::<
       (#
       do theElm.balance -> putint;
       #)
  #);
```

The next example shows a use `AccountBuffer` where only instances of `Account` or subpatterns of `Account` may be inserted:

```
(# Joe: @Person;
   myAccount: @Account
   Q: @AccountBuffer;
   S: ^Account
do myAccount[] -> Q.put;
   Joe[] -> Q.put;
   B.get -> X[];
   myAccount[] -> Q.put;
   Q.get -> S[];
#)
```

Note! More explanation here regarding Q.get?

In addition the type of elements retrieved from `Q` is known to be of type `Account`

6.1 First version of mapping

Note! Better heading needed

The pattern `Buffer` is mapped into the following Java class. The operations `put` and `get` are mapped into a method and an inner class following the general mapping for nested patterns.

The special thing here is the treatment of the virtual pattern `elm`. Since `elm` is declared as a virtual `object`, the type of `theElm`, the argument `a` of `put` and the argument `a` of the enter-method of class `put` all become `Object`.[2]

```
class Buffer extends Object {
  Object theElm;

  void put(Object a) {
     put I = new put();
     I.enter(a);
     I.do
    }
  Object get() {
     get G = new get();
```

[2]There is a subtle difference between `object` in BETA and `Object` in Java that we will ignore in this paper

```
      G.do();
      return G.exit();
   }

   class put {
      Object E;
      void enter(Object a) { E = a; }
      void do() { theElm = E; }
   }
   class get {
      Object X;
      void do() { X = theElm; }
      Object exit() { return X; }
   }
}
```

The BETA example above of using Buffer is straight forward to map into the following CJ program:

```
Account B;
Person Joe = new Person();
Object X;
Account myAccount = new Account();
Account S;
B.put(myAccount);
B.put(Joe);
X = B.get();
```

The pattern `AccountBuffer` is mapped into the following Java class:

```
class AccountBuffer extends Buffer {
}
```

As can bee seen, the body of the class is empty[3] and `AccountBuffer`, the reason being that the body of the BETA `AccountBuffer` only extends the virtual pattern `elm` to be at least `Account`. This extension cannot be expressed in Java. Instead we have to make use of casts at various places. The BETA example of using `AccountBuffer` thus maps into:

```
Person Joe = new Person();
Account myAccount = new Account();
AccountBuffer Q;
Account S;

Q.put(myAccount);
S = (Account) Q.get();    // cast
```

[3]This is not entirely true since the current version of JBeta generates dummy-methods corresponding to `elm` in `Buffer`

Note the statement `S = (Account) Q.get();` where the value returned by `Q.get()` must be casted to `Account`. This cast is superfluous since the BETA compiler knows that it will not be needed. This can as mentioned not be expressed in Java.

Note! Covariance should be mentioned

6.2 Mapping `display` into CJ

The mapping of the virtual pattern `display` follow the general scheme of mapping a nested pattern: a method and an inner class is generatated as shown below:

```
class Buffer extends Object {
  ...

  void display() {
     display D = new display();
     D.do();
   }

  class display extends Object {
     void do() {
       System.out.println(``Buffer: ``);
       do_1();
     }
     void do_1();
  }
}
```

In `AccountBuffer`, the extension of `display` gives rise to a redefinition of the `display`-method and -class:

```
class AccountBuffer extends Buffer {
  ...

  void display() {
     display D = new display();
     D.do();
   }

  class display extends Buffer.display {
    void do_1() { System.out.println(theElm.balance); }
  }
}
```

As can be seen a new `display`-class is defined in `AccountBuffer` and this class is a subclass of the `display`-class from `Buffer`. The `display`-method generates an instance of the `display`-class from `AccountBuffer`. In

```
    Buffer B = new Buffer();
    B.display;
    Buffer A = new AccountBuffer();
    A.display()
```

`B.display()` executes `display` as defined in `Buffer`. `A.display` executes `display` as defined in `AccountBuffer`. This results in execution of the `do`-part of `Buffer.display` followed by execution of the `do`-part of `AccountBuffer.display` in accordance with the BETA semantics of method combination using inner.

6.3 Mapping nested virtual classes

A virtual pattern may also be a nested pattern as shown in the next example:

```
Structure:
  (# root: @Node;
     Node:<
       (# label: @integer;
          insert:< (# N: ^Node enter N[] do inner #);
       #);
     add:
       (# V: @integer; N: ^Node
       enter V
       do &Node[] -> N[];
          V -> N.label;
          N[] -> root.add
       #)
  #);
```

An instance of pattern `Structure` has three attributes: an instance variables `root` of type `Node`, a nested virtual pattern `Node`, and an operation `add` for adding a `Node` to the `structure`. A `Node` has two attributes: an instance variable `label` and an operation `add`.

Since `node` is a nested virtual pattern, `Node` may be extended in subpatterns of `Structure`. Instances of `Node` will then be instances of the extended pattern. In `Structure` two instances of `Node` are created: one by `root: @Node` and one in `add` by the statement `&Node[] -> N[]`. These instance will be of a possibly extension of `Node`.

In pattern `List`, `Node` is extended with an instance variable `succ` and an extension of `insert`.

```
List: Structure
  (#
     Node::<
       (# succ: ^Node;
          insert::< (# ... #)
       #)
  #);
```

The extension of the virtual `insert` is supposed to append the enter-argument `N` to the list defined by `succ`.

If `L` is a `List` then `L.root` and the instances created by `L.add` below are instance of the extended `Node` pattern.

```
(# L: @List;
do (for i: 10 repeat i -> L.add for);
#)
```

The next example shows another example of an extension of `Node` in a sub-pattern `BinTree` of `Structure`:

```
BinTree: Structure
  (#
     Node::<
       (# left,right: ^Node;
          addRight: @boolean;
          add::< (# ... #);
       #)
  #)
```

Pattern `Node` is extended with instance variables `left`, `right` and `addRight`, the latter in the extension of `add` to alternate between adding `N` to the left- or right part of the binary tree.

The example shows how to add elements to a `BinTree`.

```
(# T: @BinTree;
do (for i: 10 repeat i -> T.add for);
#)
```

6.3.1 Mapping of nested virtuals into CJ

Pattern `Structure` is mapped into the CJ class shown below. The parts relating to the nested virtual `Node`-pattern is the method`Node new$Node()` and the inner `class Node`. For simplicity the `add`-methods of `Structure` are shown as CJ-methods although they are really mapped into a method and a class:

```
   class Structure {
       Node root = new$Node();

       Node new$Node() { return new Node(); }

       void add(int V) {
          Node N = new$Node();
          N.label = V;
          root.add(N);
       }

      class Node {
```

```
      int label;
      void add(Node N){ ... };
   }
}
```

Pattern `List` is mapped into the following CJ class:

```
class List extends Structure {

    Structure.Node new$Node() { return new List.Node(); }

    class Node extends Structure.Node {
       Structure.Node succ;
       void add(Structure.Node N) { ... }

    }
}
```

> *Note! We use* `Structure.Node` *above, to ensure that* `new$Node()` *is overwritten. Could we use* `Node` *instead? Return type is not used to distinguish signatures?*

Here it should be noticed that a an inner class `Node` corresponding to the extension of `Node` in `List` is defined. This class is a subclass of the `Node` class (denoted `Structure.Node` defined in `Structure`.

Also, the method with signature `Structure.Node new$Node()` is a redefinition of the corresponding method in `Structure`. The redefinition ensures that an instance of the `List.Node` is returned in accordance with the semantics of virtual classes.

Finally we consider the use of `List`:

```
List L = new List();
for (i=0; i<=10; i++) L.add(i);
```

> *Note! Need external instances L.Node, T.Node*

6.4 Summing up

For a pattern: a call- and a new-method.

Also the case for a virtual pattern

For a non-nested (global), no inner class

For a nested virtual: both

Special dummy/empty cases that may be avoided

Extra casts are needed

Note, however, that how close CJ is to support virtual patterns. Mainly a matter of the new-method and avoiding the casts. We may simulate virtual classes by the above means.

7 Expressions and statements

Expressions and statemenst are in most cases straight forward to map intp CJ.

Basic types. BETA has basic types corresponding to 8, 16, and 32 bits signed and unsigned integers, 8, and 16-bit characters, and 64 bit floating point numbers. (, 16, and 32-bits signed number are directly supported by CJ - for the unsigned version a little extra effort is needed.

Binary and unary operators. All of the binary and unary operators of BETA maps direcly to corresponding operators in CJ.

Multiple return values. A BETA pattern may return a list of values. This can not be directly implemented in JVM/CLR – instead the list of return values are stored in instance variables and a reference to the method-invocation is returned and used subsequently to retrieve the return list.

Statements. BETA's if-, and for-statements also maps easily into CJ.

Note! Bit operations, enter/exit combination

7.1 Leave and restart

In BETA is possible to exit a nested scope of method invocations:

```
Foo:
  (#
  do ...;
     L: (# Bar:
             (#
             do ...; leave L;
                ...; restart L; ...
             #);
        do ...; Bar; ...
        #);
     Lx: ...
  #)
```

Execution of `leave L` implies that execution continues at the point of `Lx` — equivalent to `goto Lx`. Execution of `restart L` implies that execution continues at `L` — equivalent to `goto L`. Execution of `leave/restart` may take place within arbitrarely nested pattern invocations[4] — in the example shown within the pattern `Bar`.

Notice that this involves *stack unwinding*, since the stack frame for `Bar` must be removed, before execution continues at `Lx`. JVM and CLR both have bytecodes for making an arbitrary jump to another bytecode, *but only within the current method*. The stack unwinding is thus not directly possible. It can be obtained, though, by using the exception mechanism. At the point of `L`, an exception handler is set up to catch exceptions that may be thrown by `leave` and/or `restart`. `Leave` or `restart` are then implemented by throwing

[4]The patterns must be declared in the scope of `L`, i.e. within the object labelled by `L`

the appropriate exceptions. The exception thrown has to contain information about

1. The *object* to unwind to
2. Which *label* in this object is the target

At the label definition site the exception handler code must then test whether the object to unwind to is identical to itself *and* the target label is numbered identical to it's own label. If so, the handler makes an unconditional jump to the first statement after the labeled region. If both conditions are not met, the handler propegates (re-throws) the exception further up the stack.

A special problem comes with implementing leave/restart across Coroutine borders. As detailed below, the current Coroutine implementation involves Threads, and the propagation of the leave/restart exception would thus involve catching the exception in the thread's entry point, and signal this to the parent thread. This has not yet been implemented.

Although we have not yet made any performance measures of the resulting code, we expect the use of the exception mechanism to implement leave and restart to be inefficient.

7.2 Pattern variables

BETA supports patterns as first class values that can be passed as arguments to other patterns and returned as values. Pattern variables are used to denote the type of a pattern. This can be used to instantiate new objects corresponding to that pattern, and to test subpattern-relations (using relational operators) between two patterns. In that sence it resembles the `Class` class of Java and the `Type` class of .NET. However, because of the general block structure in BETA, a pattern variable cannot simply be mapped to `Class/Type`. Consider this example:

```
T: (# P: (# ... #); ... #);
X1, X2: @T;
```

The two patterns `X1.P` and `X1.P` are considered different patterns, since they have different surrounding objects. This means that the test `X1.P## = X2.P##` will return `false`. The expression `X1.P##` denotes the pattern value of `X1.P`. Thus a simple test on the `Class/Type` of the generated classes will not suffice to implement, e.g., the pattern value test mentioned – we need to somehow combine the `Class/Type` information with the surrounding object. Java has no corresponding concept. .NET has the so-called *delegates*[9], which are used to tie up a *function pointer* and a specific object. As mentioned, we need to combine the full type information of the pattern with a specific surrounding object, so we cannot use delegates. instead we define a CJ class `Structure` with the following outline (here Java syntax):

```
public class Structure {
     public Object iOrigin;
     public Class iProto;
     ...
     public static boolean ltS(Structure arg1, Structure arg2)
     {
        ... determine if arg1 is same pattern as arg2 ...
     }
     ...
}
```

As can be seen, the two fields `iOrigin` and `iProto` are used to hold the object reference to the surrounding object, and the `Class` information of the pattern, respectively. A number of operations can then be implemented, like the `ltS`, which implements the test for subpattern relation. The implementation use the reflection libraries of Java and .NET to run through the superclass chain, but takes into account the corresponding origin fields.

Class `Structure` also contains methods for instantiating objects given a pattern value. Again this makes heave use of the platform reflection libraries, which provide methods to make instances of a given class.

Given this class `Structure` it is pretty straight-forward to support the various pattern variable features of BETA, but seen from Java/C#, the code will appear strange, since pattern variables will be qualified with class `Structure` in the generated CJ code. This is one place that the mapping from BETA to CJ is less elegant, than we had hoped for.

8 Coroutines, concurrency and synchronization

BETA has semi-coroutines in the style of Simula. BETA coroutines may be executed in the usual non-premptive way as known from Simula but also in concurrency with preemptive scheduling. For synchronization of concurrent coroutines, a `Semaphore` pattern is avaliable to express synchronization at the basic level. The BETA mechanisms for coroutines, concurrency and synchronization are at a much more primitive level than in CJ. However, the abstraction mechanisms of BETA makes it easy to build higher-level concurrency mechanisms such as monitor and Ada-like rendezvous. The BETA libraries include a number of such concurrency abstractions. See [7] for examples of how to define such abstractions.

Since CJ does not support Simula-style coroutines, there is no direct way of mapping BETA coroutines to CJ. Instead BETA coroutines are implemented in CJ by means of threads. This results in a considerable overhead – basically a coroutine shift is much simpler than a thread shift.

In the following a brief introduction to Simula-like semi-coroutines is given followed by a description of the implementation in CJ. The implementation is heavily inspired by [4].

8.1 Semi-coroutines

In this section the notion of a semi-coroutine is described as an extension of CJ. The basic functionality of a coroutine is defined in a class `Coroutine` – similar to class `Thread` for implementing threads. Class `Coroutine` has the following structure:

```
class Coroutine {
  ...
  void call() { ... }
  void suspend() { ... }
  abstract void Do(); // abstract
}
```

- A `Coroutine` has a special method `Do` which defines the main action-part of the coroutine – similar to the `run`-method for `Thread`.
- A concrete coroutine is defined as a subclass of `Coroutine` including a redefinition of the `Do`-method.
- If `S` is a reference to a `Coroutine`, then `S.call()` will invoke the `Do`-method of `S`.
- Execution of a `suspend` in the `S.Do()` (or methods invoked from `S.do()`) will return to the point of `S.call()` and resume execution after `S.call()`.

A semi-coroutine is a simple form of thread. Each coroutine is a thread of method activitations, but at most one coroutine is executing at a given point in time. In a situation with concurrent threads, each such thread may contain coroutines. In this paper, we assume that a program contains at most one thread.

In the following, an example of a CJ program using coroutines is shown. The example includes two generators of integers. The first generator is defined by class `Adder` below. The constructor of an `Adder`-object takes a start value. For each activation (`call`) of an `Adder`-object, a new value is generated and returned by means of the attribute `res`. The values computed are the sequence `2 * start`, `2 * (start+1)`, ... The value is computed recursively by means of `compute`. When a new value has been computed, `compute` executes a `suspend`. At subsequent calls, `compute` is called recursively to compute the next value.

```
class Adder extends Coroutine {
   public int res;
   int start;
   public Adder(int s) { start = s; }
   void compute(int V) {
      res = 2 * V;
      A1:
      suspend();
      A2:
            compute(V+1);
        }
        public void Do() { compute(start); }
    }
```

Class `Multiplier` is similar to class `Adder`. The only difference is that the values computed are `start * start`, `(start+1) * (start+1)`, ...

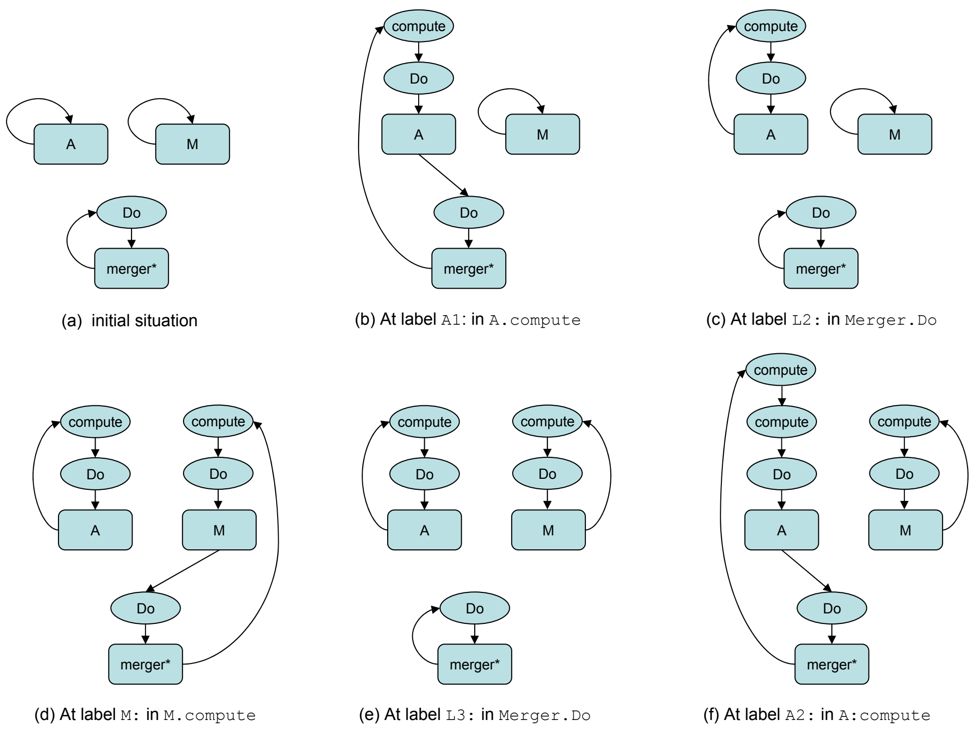
```
    class Multiplier extends Coroutine {
        public int res;
        int start;
        public Multiplier(int s) { start = s; }
        void compute(int V){
            res = V*V;
            M:
            suspend();
            compute(V+1);
        }
        public void Do() { compute(start); }
    }
```

Class `Merger` creates an instance of `Adder` and one of `Multiplier`. It then prints the values generated by the two objects in ascending order:

- The two first values are produced by executing `A.call()` and `M.call()`.
- The two values are then compared (`A.res < M.res`) and the smallest one is printed.
- If `A.res` is smallest, the next value from `A` is generated by executing `A.call`.
- If `M.res` is smallest, the next value from `M` is generated by executing `M.call`.
- This process continues until six values have been printed.

```
    class Merger extends Coroutine {
        Adder A = new Adder(3);
        Multiplier M = new Multiplier(2);
        public void Do() {
            int i;
            L1:
            A.call();
            L2:
            M.call();
            L3:
            for (i=0; i<6; i++){
                if (A.res < M.res) {
                    System.out.println("A: " + A.res);
                    A.call();
```

```
         }
         else {
             System.out.println("M: " + M.res);
             M.call();
         }
     }
   }
   public static void main(String args[]) {
      Merger merger = new Merger();
      merger.call();
   }
}
```



(a) initial situation

(b) At label `A1:` in `A.compute`

(c) At label `L2:` in `Merger.Do`

(d) At label `M:` in `M.compute`

(e) At label `L3:` in `Merger.Do`

(f) At label `A2:` in `A:compute`

Figure 1: Execution states of Merger coroutines

In figure 1 is shown six snapshots of the execution states of the Merger example.

- The round square boxes illustrate `Coroutine`-objects.
- The oval boxes illustrate method invocations.

- The arrows illustrate *caller-links* showing the calling structure of method invocations, i.e. the method invocation stack.
- Figure (a) shows the initial situation during execution of `merger.call()` in `main` at the label `L1:` of `merger.Do`.
  - The coroutines `A` and `M` have been generated.
  - By convention, a coroutine associated with a running thread is marked by a `*`. In this example the `merger`-coroutine is associated with a a running thread. The caller-link of the active coroutine refers to the method currently being executed. In (a), `merger*` refers to the `Do`-method which is currently executing.
  - For suspended coroutines, the caller-link refers to the top element of the execution stack of the coroutine. In the initial situation after a coroutine has been generated, the caller-link refers to the coroutine object itself. This is the situation in figure (a) where the caller-link of `A` and `M` refers to the `A`- and `M`-objects respectively.
- Figure (b) shows the situation where the first `A.call()` in `merger.Do` is being executed. This implies that the `A`-coroutine is executed by `merger`. Execution of `merger` implies that the `Do`-method of `A` is invoked. The `Do`-method then calls `compute`. The snapshot is at the label `A1:` just before `suspend` in this first invocation of `compute`.
- Figure (c) shows the situation after execution of `suspend` in `A` at the label `L2:` in `merger.Do`. Execution of `suspend` in `A` implies that execution of `A` is suspended. In a suspended coroutine, the caller-reference of the coroutine (`A` in this example) refers the top element of the execution stack of the coroutine – `compute` in this example.
- Figure (d) shows the situation where the `M.call()` after the label `L2:` has been executed by `merger`. This implies an execution of `M` leading to a similar situation for `M` as the one described for `A` in (b). The snapshot is at the label `M:` in `M.compute`.
- Figure (e) shows the situation at `L3:` in `merger` where `A` and `M` both are suspended.
- Figure (f) shows the situation after a subsequent call of `A` has been executed leading to a second invocation of `compute`. The snapshot is at the label `A2:`.

8.2 Implementation of class `Coroutine`

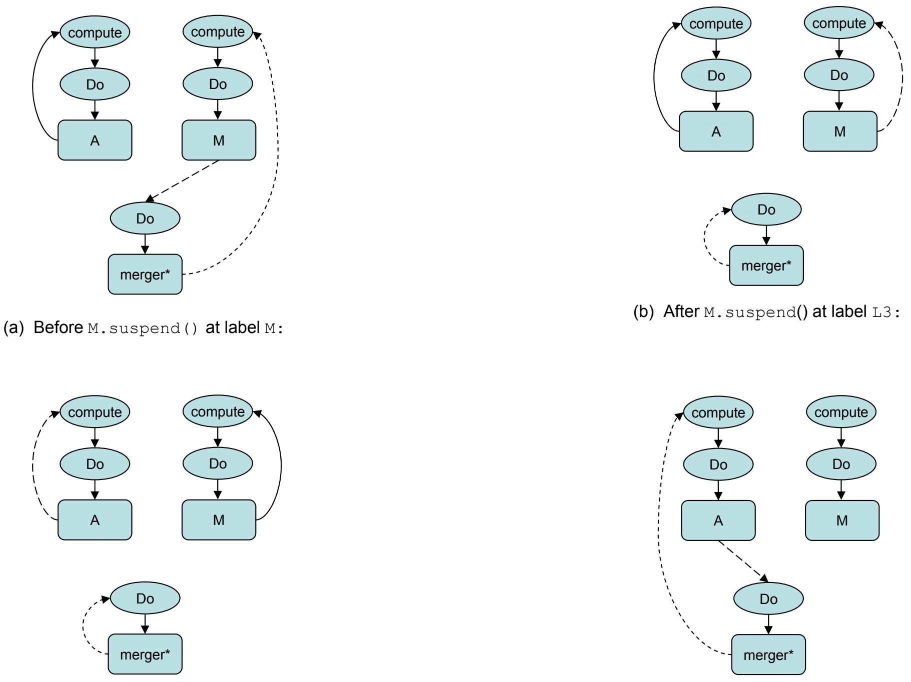
Class `Coroutine` is implemented by means of a class `Component` as shown in the following example:

```
class Coroutine {
  private Component thisC = new Component(this);

  void call() { thisC.swap(); }
  void suspend() { thisC.swap(); }
  abstract void Do();
}
```

As can be seen, the implementations of `call` and `suspend` are identical and the magic is in the `swap`-method.

A coroutine shift (call or suspend) is basically just a *swap* of two references. This is illustrated in Figure 2



(a) Before `M.suspend()` at label `M:`

(b) After `M.suspend()` at label `L3:`

(c) Before `A.call()` at label `L3:`

(d) After `A.call()` at label `A2:`

Figure 2: Illustration of the call and suspend

- The transition from (a) to (b) illustrates a suspend of `M`. In (a) there is a caller-link from the object `M` to the method invocation `merger.Do` and a caller-link from the object `merger*` to the method invocation `M.compute`. Execution of suspend implies that these two links are swapped. The result may be seen in part (b) of the figure.

- The transition from (c) to (d) illustrates a call of `A`. In (c) there are caller-links from `merger*` to `merger.Do` and from `A` to `A.compute`. Execution of `A.call()` implies that these two links are swapped. The result may be seen on part (d) of the figure.

As can be seen it is pretty simple to implement coroutines. All that is needed is a caller-link to implement the call-stack and a simple swap-operation. The caller-link corresponds to the return link of a usual method invocation stack. The BETA-implementation of coroutines is implemented this way. Unfortunately the situation i CJ is a bit more complicated since we have to get around the CJ-run time structure and thread mechanism. This is further described in the next section.

The notion of semi-coroutine originates from Simula, which also includes the original Conway-style of symmetric coroutine [2]. The coroutine model based on swap originates from Dahl & and Wang [3]. BETA has adapted the semi-coroutine model as described in [7], chapter 13 where it is also described how to implement symmetric coroutines using semi-coroutines.

8.3 Class `Component`

In this section we describe the details of the coroutine implementation in CJ. Class `Component` is as mentioned implemented using threads and could be defined as a subclass of class `Thread`. For reasons of efficiency – see later – the thread-part of a `Component` is defined in an associated `Runner`-object. Class `Runner` is a subclass of class `Thread`. Class `Component` has the following structure:[5]

```
class Component {
   static Component current;     // The current executing Component
   private Component caller;     // The calling Component

   private Runner myRunner;      // the associated Thread
   private Coroutine body;       // The actual Coroutine
   private boolean isTerminated; // True if terminated

   public Component(Coroutine b) {
      body = b;
      caller = this;
      myRunner = new Runner(this);
   }

   public void swap() {
      Component old_current = current;
      // swap pointers
```

[5]In this CJ code, the synchronization mechanisms of JAVA is used. JAVA defines the `wait()` and `notify()` methods directly on `java.lang.Object`. For .NET this is not the case; instead the .NET class library supplies the `System.Threading.Monitor.Wait(anObject)` and `System.Threading.Monitor.Pulse(anObject)`, respectively

```
         current = caller;
         caller = old_current;

         synchronized(old_current.myRunner) {
            // start or resume new current
            current.myRunner.go();
            // terminate or suspend old current
            if (old_current.isTerminated) {
               return;
            } else {
              try{ old_current.myRunner.wait(); }
              catch (InterruptedException e) {}
            }
         }
      }
   }
```

The current active Component of the coroutine-system is represented by the static field `current`.

When an instance of `Component` is created the constructor sets up the initial situation where:

- The `body`-field refers the `Corutine`-object, implemented by this `Component`
- The `caller`-field refers to this `Component`
- A `Runner`-object is created and assigned to `myRunner`.

When a `call` or `suspend` is executed by a `Coroutine` the `swap`-method of the associated `Component` is executed.

- The first part of `swap` implements a swap of `current` and `caller` as described above. ...
- Then swap invokes `current.myRunner.go()` which starts the Thread if this is the first call of swap/call. Otherwise it resumes execution of the new active `Component` – the one referred by current. Since `swap` implements `call` as well as `suspend` there are basically two situations:
  - **call**: If `current` has not been called before, the `Do`-method of the associated `Coroutine` is executed, see implementation of `Runner.go()` below. If `current` has been called before, it has executed a `suspend`, execution is resumed after this `suspend`.
  - **suspend**: Execution is resumed after the invocation of `call`.

Below is shown the implementation of class `Runner`.

```
   class Runner extends Thread {
      Component myComponent;
   Runner(Component C) { myComponent = C; }

   public void run() {
      myComponent.body.Do(); // Run Do-method of Coroutine

      // Terminate this Component
      myComponent.isTerminated = true;
      myComponent.swap();
   }

   public void go() {
     if (!isAlive()) {
        start(); // calls run
     } else {
        synchronized(this) {
          notify(); // resume this
        }
     }
   }
}
```

In Figure 3 is shown how the logical structure of a coroutine (part (a)) is represented using the classes `Component` and `Runner` (part (b)). The `merger`-coroutine has executed a call of `A`. The `merger`-coroutine is waiting in `old_current.wait()`, i.e. in `merger.myComponent.myRunner.wait()`.

Note! Peter: Ændret fra on a synchronize on its corresponding `myRunner`*-object, også i figuren, CHECK!*

Figure 4 illustrates the representation of `A` when `A` has executed a `suspend`. `A` is now waiting on a synchronize on its corresponding `myRunner`-object. The `merger`-coroutine is currently active.

8.4 Mapping BETA coroutine

Using class Component above, it is pretty straight forward to implement BETA's coroutines.

BETA has a toplevel pattern called `Object`, which in CJ is mapped to class `BetaObject`. Class `BetaObject` has a field `comp$` referring an associated `Component`.

```
class BetaObject {
   Component comp$
}
```

Note! Peter: This class corresponds to the `Coroutine` *class defined above for CJ, with* `Comp$` *corresponding to* `thisC`*. In fact, the CJ code above could be used unchanged for BETA if* `BetaObject` *was defined as a subclass of* `Coroutine`*, but for efficiency reasons*

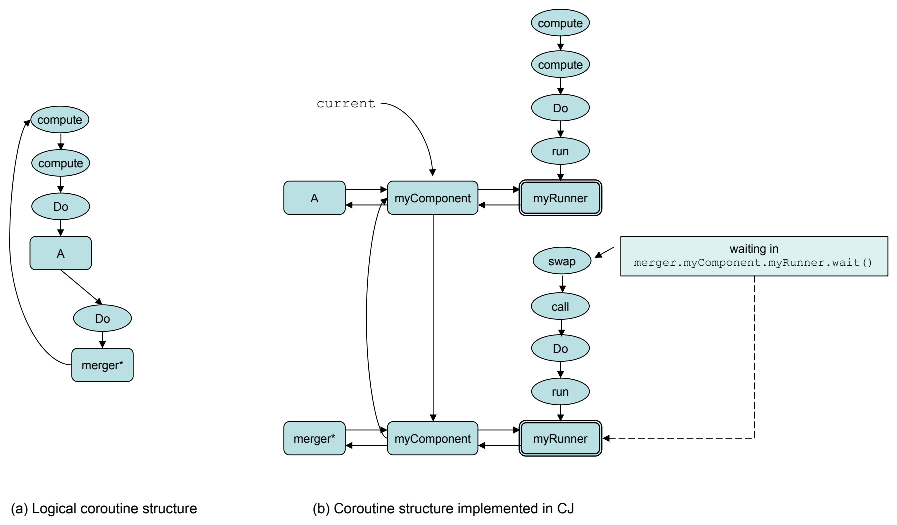


Figure 3: Illustration of coroutine implementation in CJ

> *the BETA implementation use a version of class* `Component` *where* `BetaObject` *is used instead of* `Coroutine`

Consider the following example of a BETA coroutine:

```
MyCo:
   (# foo:
       (#
       do ...; suspend; ...
       #)
   do ...; foo; ...
   #);
 M: ^| MyCo;

 &|MyCo[] -> M[];
 L: M;     (* a call of M *)
```

The `|` in `M: ^| MyCo` specifies that `M` is a reference to a coroutine of type `MyCo`. Similarly, the `|` in `&|MyCo` specifies that a coroutine instance of `MyCo` is generated.

> *Note! Peter: Notice that this means that* any pattern *may be used as a Coroutine - this is the reason for having the* `Comp$` *reference in each* `BetaObject`

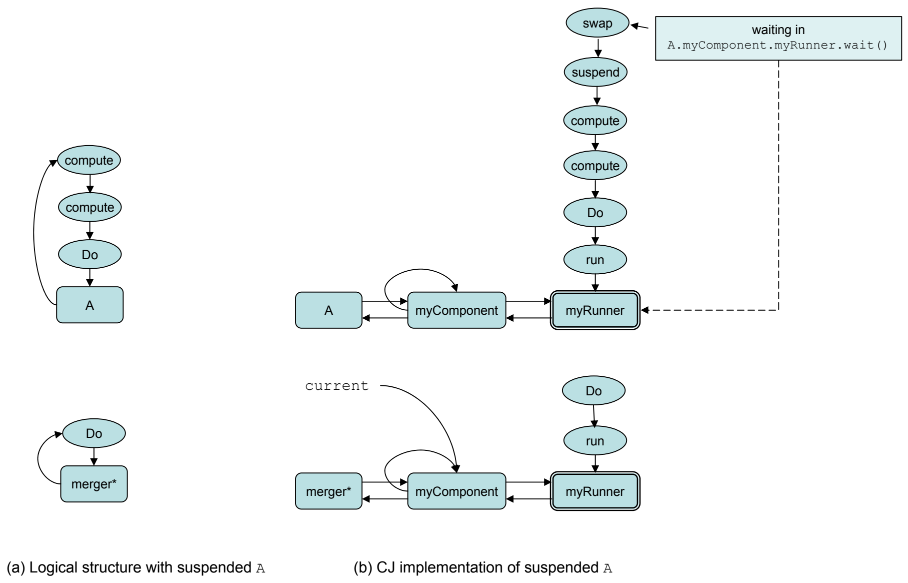


Figure 4: Illustration of suspended coroutine implementation in CJ

A coroutine call in BETA has the same syntax as invoking an object. A call of `M` takes place at the label `L`.

A BETA coroutine is mapped into CJ in the following way:

- **Generation**. When an instance of a coroutine is generated, a corresponding instance of `Component` is generated and assigned to the `comp$`-field.
- **Call.** A call `M` is mapped into `M.comp$.swap()`.
- **Suspend.** A suspend executed by `M` is similarly mapped into `comp$.swap()`.

Note! Perhaps a figure showing M and a reference to the associated Component. Peter: basically the same figure as figure 3, with BetaObject instead of Couritine

Note! Check this

8.5 Remaining details of the implementation

As mentioned the mapping of coroutines is heavily inspired by [4], which describes a complete implentation of both semi-coroutines and symmetric coroutines from Simula. Since BETA coroutines are simpler than Simula coroutines the mapping to CJ is aslo simpler than the one described in [4]. Following [4] the actual implementation of coroutines is further refined as described below:

- A Java program does not terminate as long as there are active threads. When a BETA program terminates, there may be suspended coroutines that have not terminated. To handle this the statement `SetDaemon(true)` is called for each Runner to indicate that the program may terminate even if the Runner has not terminated. C#?

- In the above implementation a new `Thread` is generated whenever a new `Coroutine` is generated. Thread allocation is very expensive in Java. In a some Simula and BETA program a large number of coroutines may be allocated duringa program execution. Often only a few of these are alive at a given point in time. For this reason it improves efficiency to keep a pool of Threads.

As mentioned a terminated BETA program may generate a large number of coroutines. In BETA, a coroutine is alive as long as there is a reference to it form a live object. When a coroutine cannot be reached from a live object, it may be garbage collected.

This is however a problem in in the Java implementation. Even if a BETA coroutine is not reachable from an active object it is reacheable form the associated `Runner/Thread`. And since non-terminated threads are roots for the garbage collector in Java, these coroutines and associated `Runner`-objects are never garbage collected. This is currently and unsolved problem for Java.

For C#?

The complete implementation of class `Component` and `Runner` is shown in the appendix.

9 Evaluation

9.1 Language interoperability

It is simple and straight forward to use CJ classes from BETA. This incluses instatiating CJ classes and and invoking methods on CJ-objects. It is also possible to inherit from CJ classes in BETA.

To be able to use a CJ class from BETA a BETA external class corresponding to the CJ class must be defined. Since clr/jvm is name & type-based, one just need to specify the signature of the methods that will beused from BETA. This is much simpler than using COM where the full interface has to be specified since the interface is based on offsets in the vtable. This means that all methods preeceding a used method in the vtable must be declared.

In most cases, however, the external class may be generated from the classfile which means that the programmer does not have to woory about specifying the external class.

The use of BETA pattersn from CJ is just as simple and straight forward. Classes and methdos generated from BETA may be used in CJ. This includes inheritance from BETA classes. A CJ programmer must of course know the mapping of BETA to CJ. Currently there is no tool forgenerating a CJ interface to BETA. it is possible to generated CJ code form the class-files but this is not optimal. We discuss the mapping further below.

There are some remainingissues regarding interoperability due to incompatibility of BETA and CJ. These are

- Issues with constructors, including default constructor
- BETA object requires surrounding object (origin)
- Overloading
- Issues – Libraries and frameworks in Java/C# – Java-string, C#-String, BETA-text

  Automatic coercion implemented
- External class interface

  Interface syntax needed – currently clumsy

  External name & location

  Automatic include of interface files

  .NET assemblies

  Java class-files

Note! A summary of the experiences with language interoperability on JVM/CLR

9.2 Quality of the mapping and changes to BETA

Given the generality of BETA compared to CJ, we find that the current mapping to CJ is satisfactory from a usability point-of-view. The main nuisance is that two methods and an inner class is generated for each pattern attribute.

When the BETA project was initiated the main goal of unifying abstraction mechanisms like class and method into patterns was to obtain a uniform treatment of all abstraction mechanisms of the language. Given a pattern abstraction it was expected that specialized patterns for class and methods should be available. This was termed *language restriction*. However, in practice there was no real demand/need for these special purpose abstractions.

In the context of mapping BETA to CJ, the idea of language restrictions have been reintroduced. It is now in fact possible to annotate a pattern with either `class` or `proc` meaning that the pattern can only be used as a class or method. For such annotated patterns, only one method needs to be generated. And for `proc`-patterns the inner class may in fact be inlined – the latter optimization has not been implemented.

At this stage of the project, efficiency and optimization of the mapping has had little attention.The first impression, however, is that the JVM/CLR implementation results in code that is significantly slower than the native BETA compilers. We do expect to able to improve on parts of the implementation, but certain parts such as leave/restart and coroutines the mapping appear to be inherently inefficient.

9.3 Platform issues

- No static link on stack frames
- Nested procedures/methods cannot be implemented using the stack
- Very rigid typing of class-fields
- And method-locals on .NET
- Constructor initialization
- Impossible to make setups before calling super constructor
- General exit out of nested method calls
- No support for covariant arguments
- No support for covariant return types
- Active objects
- .NET class references must be fully qualified, including location of binding
- Problems with separate compilation
- Large number of classes generated due to the generality of BETA patterns
- Java class files can only contain one (non-static) class per file
- Lots of class files are generated
- .NET assemblies can contain any number of classes
- No swap and dup_x1/x2 on .NET
- Requirements for fully typed local variables in .NET.
- Means that swap cannot be implemented in backend using local variables unless type of two top-of-stack objects are known.
- Type of field: super/this class

9.4 Platform advantages

Well-defined run-time format

More than just a calling convention for procedures

Memory management

Storage allocation

Object format & method activation format

Garbage collection

Efficient code generation

We will have BETA for all Java- and .NET platforms

Can this be utilized efficiently?

If simple and direct mapping - yes

If complex mapping - no?

Inlining may help

Can rely on generating methods that the JIT inlines

9.5 Conclusion

It has been possible and in most cases straight forward to map BETA to CJ. The main problems have been coroutines, leave/restart and nested procedure patterns.

It has been possible to find a redable mapping to be used by CJ programmers using BETA.

Language incompatabilities...

The jvm/clr platforms .....

More generality of future platforms ....

But language interoperability possible to a certain degree. However, is it just a matter of different syntax? What is the pincipal difference between a mschineplatform like Intel and jvm/clr?

Appendix A. Coroutines

In this appendix, the classes `Component` and `Runner` are shown with the code for handling the thread-pool.

```
class Component {
   static Component current;     // The current executing Component
   private Component caller;     // The calling Component

   private static Runner firstFree;

   private Runner myRunner;      // the associated Runner
   private Coroutine body;       // The actual Coroutine
   private boolean isTerminated; // True if terminated

   public Component(Coroutine b) {
      body = b;
      caller = this;
      if (firstFree == null) {
         myRunner = new Runner();
      } else { // get at Runner from the thread-pool
         myRunner = firstFree;
         firstFree = firstFree.next;
         myRunner.myComponent = this;
      }
   }


   public void swap() {
      Component  old_current = current;
      current = caller;
      caller = old_current;

      synchronized(old_current.myRunner) {
         current.myRunner.go();
         if (old_current.isTerminated) {
            return;
         } else {
           try{ old_current.myRunner.wait(); }
           catch (InterruptedException e) {}
         }
      }
   }
}

class Runner extends Thread {
   Component myComponent;
   Runner next;'

   Runner(Component C) {
      myComponent = C;
      setDaemon(true)
   }

   public void run() {
      while (true) {
         myComponent.body.Do();
         myComponent.isTerminated = true;
         myComponent.swap();

         // return this Runner to the thread-pool
         next = firstFree;
         firstFree = this;
         myComponent = null;

         // wait for a new Component to use this Runner
         synchronized (this) {
            try{ wait(); } catch (InterruptedException e){}
         }
      }
   }
   public void go() {
     if !isAlive() { start(); }
     else {
        synchronized(this) {notify();};
     }
   }
}
```

References

[1] Andersen, P., Enevoldsen, M.B., Madsen, O.L: *Integration of BETA with Eclipse – an exercise in language interoperability*, submitted for publication

[2] Conway: Coroutines, ...

[3] Dahl,O.-J., Wang, A.: Coroutine Sequencing in a Block- Structured Environment, BIT, ...

[4] K. Helsgaun: *Discrete Event Simulation in Java*, Writings in Computer Science, Roskilde University, Denmark, 2000, http://www.dat.ruc.dk/~keld/research/JAVASIMULATION/JAVASIMULATION-1.0/docs/Report.pdf

[5] J.L. Knudsen: A Static Exception Handling Mechanism? ...

[6] O.L. Madsen, B. Møller-Pedersen: Virtual Classes - A Powerful Mechanism in Object-Oriented Programming In: Proceedings of OOPSLA'89, Object-Oriented Programming Systems, Languages and Applications, Sigplan Notices, 1989

[7] O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Object-Oriented Programming in the BETA Language. ACM Press/Addison Wesley, 1993. Out of print – a copy can be downloaded from http://www.mjolner.dk

Note! Here we should refer the new home page of BETA

[8] K. K. Thorup: Virtual classes in Java, ECOOP'97, Jyvaskala, 1997.

[9] C# Language Specification: *Delegates*

http://msdn.microsoft.com/library/en-us/csspec/html/vclrfcsharpspec_15.asp