MIA 99-42: The Fragment System: Further Specification

Table of Contents

Copyright Notice

Mjølner Informatics Report
MIA 99-42
August 1999

The Fragment System: Further Specification

Mjølner Informatics Report, MIA 99-42, August 1999

1. Introduction

The Mjølner System is based on the notion of *fragment*. The fragment system must be used for splitting a large program into smaller units (fragments). The fragment system is used to support modularization, separation of interface and implementation parts, variant control and separate compilation. It is highly recommended to use the fragment system, since this may improve the structure of the program.

The principles of the fragment system are described in [MMN93,Chapter 17]. Chapter 17 of [MMN93] is also published in [Knudsen 94]. In the following it is assumed that the reader is familiar with this description.

The description in [MMN93] is slightly more idealized than the actual implementation in the Mjølner System: In [MMN93], the syntax of the fragment language is given in terms of diagrams. The fragment language implemented by the Mjølner System has a textual syntax. In this paper, the textual syntax corresponding to the diagrams in [MMN93] is presented.

The description of the fragment system in [MMN93] is further specified, including a description of implementation restrictions compared to [MMN93].

2. Restrictions The following restrictions apply for the implementation of the fragment system:

- In the Mjølner System, slots have been implemented for the syntactic categories:
  `<<objectDescriptor>>`
  `<<mainPart>>`
  `<<doPart>>`
  `<<attributes>>`
- The alias `descriptor` can be used instead of `objectDescriptor`.
- A fragment form of the category `<<attributes>>`, may only contain pattern declarations. It cannot contain any other kind of declarations, including virtual pattern declarations, virtual pattern bindings, static or dynamic declarations.
- In the current system, fragments are organized in groups. A group is stored as a file. The BETA compiler accepts a BETA program in the form of one or more files. Each file must contain a group of fragments (i.e. one or more fragments).
- A pattern where the object descriptor is described as a slot cannot be used as a super-pattern. I.e. the following is illegal:

  ```
  A: <<SLOT Pdesc: descriptor>>;
  B: A(# ... #);  (* illegal *)
  ```

  Instead the following can often be used:

  ```
  C: (# do <<SLOT Pdesc: descriptor>> #)
  D: C(# ... #); (* legal *)
  ```

3. Fragment Language Syntax

In the following some of the examples of fragments using the diagrammatic syntax from [MMN93] will be given followed by the syntax used by the Mjølner System. The first example shows the simplest possible BETA fragment-group:

```
NAME 'mini1'ORIGIN 'betaenv'PROGRAM: descriptor(# do 'Hello world!' -> PutLine #)
```

The fragment-group is stored in the file `mini1.bet`, which is also the name of the fragment-group. The following syntax is is used by the Mjølner System:

```
ORIGIN '~beta/basiclib/betaenv'
-- program: descriptor --
(#
do 'Hello world!'->PutLine
#)
```

The origin betaenv has been expanded into a complete file name for betaenv.

The next example is an example defining a library fragment:

```
NAME 'mylib' ORIGIN 'betaenv' LIB: attributes Hello: (# do 'Hello' -> PutText #);
World: (# do 'World' -> PutText #)
```

This fragment is stored in a file `mylib.bet` and the corresponding syntax in the Mjølner System is:

```
ORIGIN '~beta/basiclib/betaenv'
-- LIB: attributes --
Hello: (# do 'Hello' -> PutText #);
World: (# do 'World' -> PutText #)
```

The following fragments is an example of a fragment including the library: `mylib.bet`

```
NAME 'mini2' ORIGIN 'betaenv' INCLUDE 'mylib' PROGRAM: descriptor (# do
Hello; World; newLine #)
```

This fragment is stored in a file `mini2.bet` and has the following syntax:

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE 'mylib';
-- program: descriptor --
(#
do Hello; World; newLine
#)
```

The following example shows a fragment with a body:

NAME 'textlib' ORIGIN 'betaenv' INCLUDE 'mylib' LIB: attributes SpreadText: {A blank is inserted between all chars in the text 'T'} (# T: @text> enter T <<SLOT SpreadText:DoPart>> exit T #); BreakIntoLines: {'T' refers to a text which is to be split into lines.} {'w' is the width of the lines.} (# T: ^ Text; w: @ Integer enter(T[],w) <<SLOT BreakIntoLines: DoPart>> #)

It is stored in a file `textlib.bet` and has the following syntax:

```
ORIGIN '~beta/basiclib/betaenv';
BODY 'textlibbody';
---LIB: attributes---
SpreadText:
  (* A blank is inserted between all chars in the text 'T' *)
  (# T: @text
  enter T
  <<SLOT SpreadText: DoPart>>
  exit T
  #);
BreakIntoLines:
  (* 'T' refers to the text to be split into lines. *)
  (* 'w' is the width of the lines. *)
  (# T: ^ Text; w: @ Integer
  enter(T[],w)
  <<SLOT BreakIntoLines: DoPart>>
  #)
```

The body of textlib is shown in the next example:

NAME 'textlibbody' ORIGIN 'textlib' SpreadText: DoPart do (# L: @integer do (for i: (T.length->L)-1 repeat (' ',L-i+1) -> T.InsertCh for) #) BreakIntoLines: DoPart do T.scan (# sepInx,i,l: @integer; do i+1->i; l+1->l; (if (ch<=' ') then i->sepInx if); (if l=w then (nl,sepInx)->T.InxPut; i-sepInx->l if); #); T.newline;

This fragment is stored in a file `textlibbody.bet`. The corresponding syntax is:

```
ORIGIN 'textlib'
-- Spreadtext: DoPart --
do (# L: @Integer
   do ...
   #)
--BreakIntoLines: DoPart --
do ...
```

Notice, that when local variables are needed in a `DoPart` slot, it may be necessary to make an inserted item in the `DoPart`. Alternatively a Private descriptor slot may be declared in the interface, and the L attribute moved to the Private fragment, which should then be placed in textlibbody.bet too.

Finally a general outline of a fragment group with several include, body and fragments is shown in the next example:

NAME F ORIGIN G INCLUDE A1 INCLUDE A2 ... INCLUDE Am BODY B1 BODY B2 ... BODY Bk F1: S1 ff1 F2: S2 ff2 ... Fn: Sn ffn

This fragment group is stored in a file `F.bet` and the syntax is:

```
ORIGIN 'G';
INCLUDE 'A1' 'A2'... 'Am;
BODY 'B1' 'B2' ...  'Bk';
Prop1; Prop2; ... Propl
-- F1: S1 --
ff1
-- F2: S2 --
ff2
...
-- Fn: Sn --
ffn
```

`Prop1, Prop2, ..., Propl` are *properties* that may be defined for a fragment. Formally the `ORIGIN, INCLUDE`, and `BODY` parts are also properties. In section 5 a list of possible properties is given.

4. Fragment Denotations

In the examples above, terms like `'~beta/basic_lib/mdev'`

were used. Below we will use the term FragmentDenotation for the 'fragment path' given in, e.g., the `INCLUDE` property. The other properties, that accept FragmentDenotations as arguments are explained in section 5.

Notice that a FragmentDenotation is *not* the same as a file name, although it resembles a UNIX file path, and although it normally corresponds directly to a (set of) file(s):

1. The separator in the FragmentDenotation is always the `'/'` character. Although this is a UNIX convention, the `'/'` must also be used on Macintosh and Windows.
2. As explained in section 3, the notation `'~beta'` is legal in FragmentDenotations on all platforms, and simply means 'the place BETA is installed'. As mentioned, the meaning of `'~beta'` can be controlled by using the BETALIB environment variable, please consult [MIA 99-36] for details.
3. The notation `'.'` means `'current directory/folder'` on all platforms, and the notation `'..'` means `'father directory/folder'`, i.e. the directory containing a given directory.
4. It is not allowed to specify an extension (e.g. `'.bet'` or `'.ast'`) in a FragmentDenotation.

There are some restrictions in the legal fragment *file* names, which also apply to the FragmentDenotations, please see section 9.

5. Fragment Properties

The fragment system allows arbitrary properties to be associated with fragments. The BETA compiler recognizes the following properties: For most users, only `ORIGIN`, `INCLUDE`, and `BODY` are relevant.

ORIGIN <*TextConst*>

The origin of a fragment is a fragment which is used when binding fragment-forms to slots.

INCLUDE <*StringList*>

Specifies one or more fragments that are always included when using this fragment.

BODY <*StringList*>

Specifies one or more fragments that fills the slots in this fragment file, but are not visible.

MDBODY <*MachineSpecificationList*>

Specifies one or more machine dependent fragments that fills the slots in this fragment file dependent on the machine type. See section 9 for further description.

BUILD <*MachineSpecificationList*>

The `BUILD` Property is used to specify rules for keeping external (i.e. non-BETA) sources up to date, and to include the external files in the link directive. The `BUILD` property unifies the `OBJFILE` and `MAKE` properties.

OBJFILE <*MachineSpecificationList*>

The object file is included in the linker-directive. This is typically an External library which is interfaced to via the External interface described in [MIA90-8]. See also BUILD and section 9.

BETARUN <*MachineSpecificationList*>

The standard BETA run-time system is replaced with the one in the object-file. See also section 9.

MAKE <*MachineSpecificationList*>

Specifies one or more makefiles to be executed before linking. See also section 9. The Makefile is executed relative to the directory, where the file containing the MAKE property is placed. See also BUILD

RESOURCE <*MachineSpecificationList*>

Specifies one or more resource files to be included in the applicatiiton. Only used on Macintosh and Windows NT platforms. See also section 9.

LIBFILE <*MachineSpecificationList*>

Is similar to OBJFILE, but specifies inclusion of a library. See also section 9.

LINKOPT <*MachineSpecificationList*>

Machine dependent options to append to link directive for programs using the fragment. Only used on UNIX platforms. See also section 9.

The terms `<MachineSpecificationList>`, `<StringList>`, and `<TextConst>` are syntactically explained in the grammar:

- Property Grammar

6. Modularization of Data Structures

This section gives some advices that can be used when modularizing data structures. Consider the following program library (`stack.bet`):

```
ORIGIN '~beta/basiclib/betaenv'
--- Lib: attributes ---
stack:
  (# element:< object;
     A: [100] ^element;
     top: @integer;
     push:
       (# e: ^element;
       enter e[]
       do top+1->top;
          e[] -> A[top][];
       #);
     pop:
       (# e: ^element;
       do A[top][] -> e[];
          top-1->top;
       exit e[]
       #);
     top:
       (# e: ^element;
       do A[top][]->e[];
       exit e[]
       #);
  #)
```

If we want to separate the interface and the implementation, this can be modularized in the following way:

Introduce the following SLOTs:

```
ORIGIN '~beta/basiclib/betaenv';
BODY 'stackImpl'
--- Lib: attributes ---
stack:
  (# element:< object;
     private: @<<SLOT private: descriptor>>;
     push:
      (# e: ^element;
      enter e[]
      <<SLOT pushBody: DoPart>>
      #);
     pop:
      (# e: ^element;
      <<SLOT popBody: DoPart>>
      exit e[]
      #);
     top:
      (# e: ^element;
      <<SLOT topBody: DoPart>>
      exit e[]
      #);
  #)
```

Create a new fragment file stackImpl.bet:

```
ORIGIN 'stack';
-- private: descriptor --
(# A: [100] ^element;
   top: @integer;
```

```
#)
-- pushBody: DoPart --
do private.top+1->private.top;
   e[] -> private.A[private.top][];
-- popBody: DoPart --
do private.A[private.top][] -> e[];
   private.top-1->private.top;
-- topBody: DoPart --
do private.A[private.top][]->e[]
```

The reason why the data representation (`A` and `Top`) is put into a descriptor slot instead of an attributes slot is that attributes slots may only contain patterns, no static items (objects) or object references. This is due to the implementation of separate compilation. Therefore it is necessary to put static items into an attribute (in this case private) that is declared by means of a descriptor slot. Because of this all accesses to the representation must be done via the private variable (see `pushBody`, `popBody` and `topBody`). Notice that the parameters are visible in the interface. If the operations had local variables they should not be shown in the interface.

7. Modularization with INNER

Programs fragments with do-parts that contain an `INNER` imperative e.g.:

```
ORIGIN '~beta/basiclib/betaenv';
--- lib: attributes ---
A: (# do imp1; imp2; INNER; imp3 #)
```

can be modularized in the following two ways depending on whether the `INNER` imperative should be visible in the interface or not.

If the `INNER` is preferred visible in the interface, the interface fragment could look like (`fooLib1.bet`):

```
ORIGIN '~beta/basiclib/betaenv';
BODY 'fooImpl1'
-- lib: attributes --
A: (#
   do <<SLOT imp12slot: descriptor>>;
      INNER;
      <<SLOT imp3slot: descriptor>>
   #)
```

and the implementation fragment (`fooImpl1.bet`):

```
ORIGIN 'fooLib1'
-- imp12slot: descriptor --
(# do imp1; imp2 #)
-- imp3slot: descriptor --
(# do imp3 #)
```

In this case a `DoPart` slot might be used instead (`fooLib2.bet`):

```
ORIGIN '~beta/basiclib/betaenv';
BODY 'fooImpl2'
-- lib: attributes --
A: (# <<SLOT imp12slot: DoPart>> #)
```

with the implementation fragment (`fooImpl2.bet`):

```
ORIGIN 'fooLib2'
-- imp12slot: DoPart --
do imp1; imp2; INNER; imp3
```

Using do-parts like this, then although the `INNER` is not visible in the interface, the A pattern may still be specialized and behave as if the `INNER` was in the interface. Notice, that when specializing a

pattern with no INNER in the do-part, the compiler will normally complain about this. But when the pattern being specialized contains a SLOT, the compiler will assume, that the SLOT contains an INNER. Thus it is possible to specialize the A pattern in foolib2.

But if the INNER imperative is placed 'inside' some structure e.g.:

```
A: (#
   do (if E1
       // E2 then INNER
       // E3 then imp
      if)
   #)
```

you might not want to show the if imperative in the interface. In this case a variant of the INNER construct may be used, in which case the interface fragment could be (fooLib3.bet):

```
ORIGIN '~beta/basiclib/betaenv';
BODY 'fooImpl3'
--- lib: attributes ---
A: (# do <<SLOT Abody: descriptor>> #);
```

and the implementation fragment (fooImpl3.bet):

```
ORIGIN 'fooLib3'
--- Abody: descriptor ---
(#
do (if E1
    // E2 then INNER A
    // E3 then imp
   if)
#)
```

If a 'normal' INNER had been used instead of INNER A, it would mean that specializations of the pattern containing the INNER in the do-part combine the actions at this point. But the pattern containing the INNER in the do-part, in this case would be the anonymous pattern in the ABody descriptor fragment. By using INNER A, it is ensured, that the control flow descents to the specialization of A although the INNER is inside the ABody descriptor.

A DoPart slot could also be used here, as in the previous example.

8. Formal Syntax of Fragment Language

The formal syntax of the BETA fragment-system is:

<TranslationUnit> ::= <Properties> <FormPart>

<FormPart> ::* <FormDef>

<FormDef> ::= -- <FormDefinition>

<FormDefinition> ::| <DescriptorForm>

| <AttributesForm>

| <DopartForm>

| <MainpartForm>

<DescriptorForm> ::= <NameDcl> : descriptor -- <ObjectDescriptor>

<AttributesForm> ::= <NameDcl> : attributes -- <Attributes>

<DopartForm> ::= <NameDcl> : doPart -- <DoPart>

<MainpartForm> ::= <NameDcl> : mainPart -- <MainPart> The grammar for `<Properties>` may be found at:

- Property Grammar

Notes:

- The symbol `--` may consist of two or more dashes (`-`).
- The old style `INCLUDE` and fragment syntax (`--INCLUDE` fragment) is no longer supported.

9. File Name Restrictions

Because of implementations details, the current version of the fragment system imposes the following restrictions on file names used for BETA programs.

1. It is not allowed for a program to use two files with the same name, say `foo.bet` (ignoring case), which both contains fragments of category `Attributes`.

2. It is not allowed for a program to use a file named, say, `foo.bet`, if `foo.bet` contains a fragment of category `Attributes`, and if there is a `SLOT` of category `ObjectDescriptor/Descriptor` or `DoPart` named `foo` in any of the files involved in the program. Again case is irrelevant.

3. It is not allowed to use the '-' (dash) character in fragment file names.

4. Because the `FragmentDenotation` separator character is `'/'` it is not allowed to use the `'/'` in fragment file names, not even on platforms where the file system would allow it.

5. In general, it is advisable to restrict the characters used in the fragment file names to be: `a-z`, `A-Z`, `0-9`, and `'_'`. If other characters are used in the fragment file names, there is a danger, that the supporting tools (such as linkers) will complain.

The symptom on breaking rule 1 or 2 is typically a '`Multiple defined symbol M1FOO`' and the like, in the linking phase, the symptom for breaking rule 3 is that the compiler or the mjolner tool ([MIA 99-39], [MIA 99-40], [MIA 99-34]) may become confused. Finally the symptom on breaking rule 5 may be a complaint from the assembler about illegal characters.

Except for rule 3, these restrictions only apply to the *file* names. The *directories / olders* containing the files, may be freely named.
A pattern where the object descriptor is described as a slot cannot be used as a super-pattern. I.e. the following is illegal:

```
A: <<SLOT Pdesc: descriptor>>;
B: P(# ... #);  (* illegal *)
```

Instead the following can often be used:

```
C: (# do <<SLOT Pdesc: descriptor>> #)
D: P(# ... #); (* legal *)
```